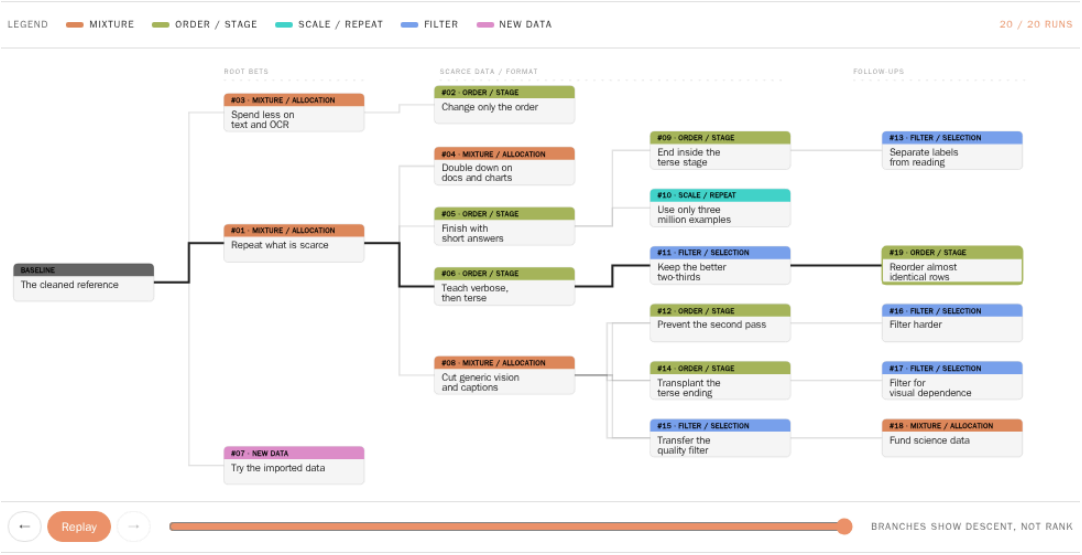
Autoresearch for Data: We Gave an Agent Our GPU Cluster to Improve Our Data Mixture



We gave an agent a budget of 11,000 H100 hours to find the best data mixture for training a 2B VLM.

AUTHORS

Nazar Drugov, Joel Niklaus, Leandro von Werra, Andrés Marafioti

AFFILIATION

Hugging Face

PUBLISHED

Aug. 26, 2026

Table of Contents

TL;DR

- We gave an agent a budget of 11,000 H100 hours to find the best data mixture for training a 2B VLM. The model, training code, evaluation, and compute budget were fixed.
- The agent's best mixture raised the aggregate score from 58.8% to 62.8%, a 6.8% relative improvement over our human-built baseline, FineVision. The baseline is the strongest mixture we found after 200K+ H100 hours of comparisons with other datasets and experiments to improve it.
- Along the way, the agent questioned our findings from the work on FineVision and ran experiments to challenge them. It also uncovered flaws in the benchmarks.

Introduction

Data is still the biggest lever for training great models. Anyone who has ever worked on data mixing will tell you that it's a grind. Your intuition is constantly challenged, brilliant ideas turn out to be useless, and in the end, the best mixture is the third one you made, where you accidentally added all your dog's photos.

Still, there is surprisingly little research into getting agents to find the best data mixture. Autoresearch is on the rise. OpenAI plans to deploy an automated research intern on 500K GPUs in September 2026! Karpathy's Autoresearch (Karpathy, 2026) and the Fast Gemma Challenge (Gemma Challenge, 2026) also show that autoresearch can work. But those projects mostly focus on code. Agents modify architectures, hyperparameters, training procedures, or inference implementations.

So why is data being left behind? How far can agents take us in data optimization?

To answer these questions, we created VLM Data Autoresearch. The agent drew its training data from FineVision, our VLM data pool, which took more than 200K H100 hours to develop. Five researchers spent months building it and optimizing its mixture. We wanted to see if the agent could find a better way to mix FineVision's datasets than we had.

In particular, we created a harness around Claude Code with the infrastructure for filtering and mixing data, importing outside datasets, annotating images with Gemma 4 31B, and submitting training runs.

This blog describes what we found.

What levers does the agent have?

The agent's task is to design the data mixture for each run: which sources the model trains on, and in what proportion. A tool then turns the mixture into the exact list of examples the run will see over its 16,050 steps.

The agent has five levers:

1. Filter. Include or exclude individual samples using properties such as length, number of images, number of conversation turns, relevance, and visual dependency.
2. Schedule. Split training into stages, each with its own mixture. For example, one recipe reserved short-answer data for the final stage. Inside a stage, the order is shuffled with a fixed seed the agent cannot set.
3. Size and repeat. Choose how many samples go into the mixture. A list longer than the run consumes is read from the top, and the tail goes unseen. A shorter list wraps around, so the model meets some examples several times.
4. Generate. Use Gemma 4 31B to annotate image-only datasets or re-annotate low-quality question-answering datasets.
5. Import. Bring in outside datasets and convert them to the format expected by the trainer.

Setting up the loop

Before the agent loop could start, we had to build everything around it. We wrote the tools the agent can use, prepared the trainer code, and trained the baseline. We then froze the files containing the training, scoring, and infrastructure code, so the agent couldn't modify them. If the agent unfreezes or modifies these files, the loop notifies us and halts.

After we started the loop, it mostly ran autonomously, except when something in the harness broke and we halted the loop to repair it.

One iteration of the loop

Whenever a training, annotation, or conversion job finishes or fails, the scheduler starts a new agent session. Each session follows the same loop:

1. The timer checks the project.

A training experiment takes about 38 hours, so keeping an agent session alive while it waits would be wasteful. Instead, `cron` runs a small polling program every 30 minutes. On each tick, it:

- starts submitted jobs when the appropriate compute slot is available;
- checks whether running jobs are still alive;
- wakes the agent if a training run, annotation job, or conversion job had finished or failed
- halts the loop if frozen code changed or 3 training runs failed consecutively.

2. The timer wakes at most 1 agent. If a job needs attention, the timer starts a fresh agent session and gives it a short message naming the job, its outcome, the location of its files, and the currently available compute. A session may run for at most 2 hours.

3. The agent reconstructs its context from files. Everything the agent needs to remember lives in files:

   - `agent.md` describes what it may change, how to submit work, and which safety rules it must follow.
   - `logbook.md`, the running research diary: what previous agents tried, how they interpreted the results, and which ideas looked worth pursuing next.
   - `log.jsonl`, the record of experiments: every completed run, its recipe, parent, chosen checkpoint, and full score curves.

   log.jsonl stores facts in a format that code can read. logbook.md stores agent's findings and decisions for us and next agents to read.

4. The agent handles the event that woke it.

   - A finished training run: it studies the score curves on all 10 benchmarks against the baseline and the parent run, picks the checkpoint that best represents the run, records the result, and writes what it learned in the logbook.
   - A finished annotation or conversion job: it reads the job report, checks how many samples survived and why the rest were dropped, and inspects the data before deciding whether to use it.
   - A failed job: it reads the SLURM logs, then either fixes and resubmits the idea or abandons it.

5. The agent chooses the next work. Based on the new evidence and the previous research record, it can write one or more training recipes, request new annotations, request a DCVLM

conversion, or spend the session analysing existing results.

6. The agent writes a handoff and exits. Before ending the session, it updates the logbook and writes its next intended direction to `state.json`.

One iteration of the loop

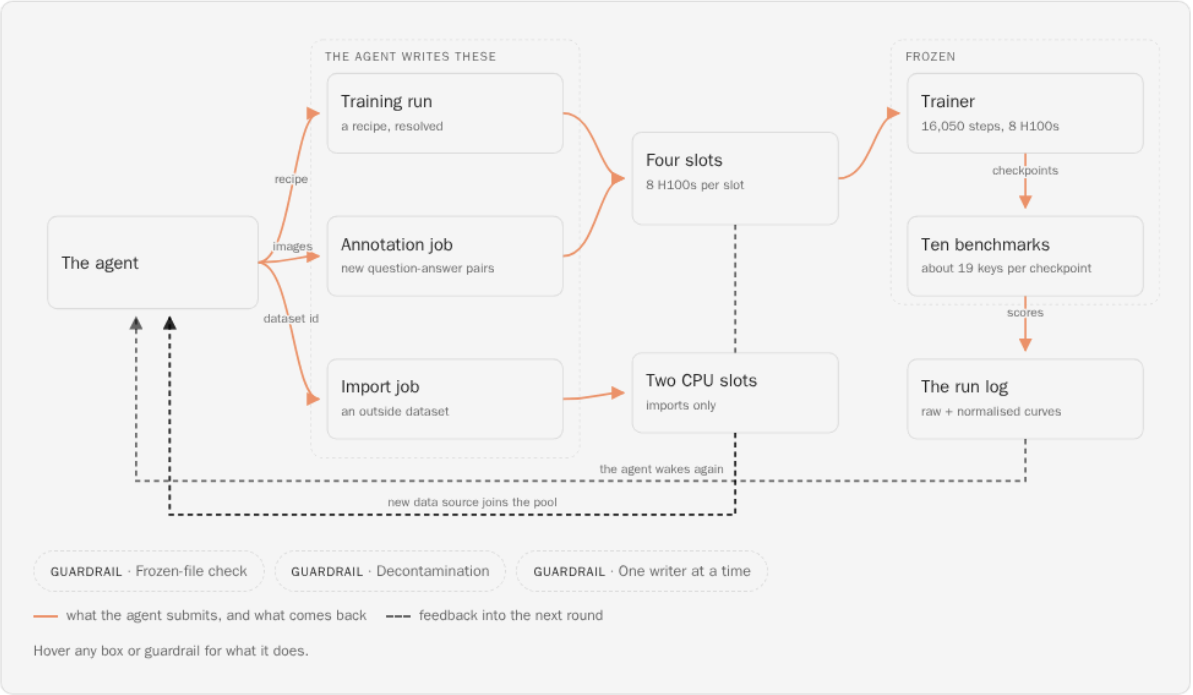


Figure 1 · Hover any box, or any guardrail, to see what it does.

Results

The agent went through 64 iterations of the loop above. It trained 19 models on different mixtures, lost 7 of its training runs to failures, and created 29 imports along the way.

Its best run improved on all ten benchmarks evaluated and raised the mean of the ten scores by 6.8% relative to the baseline. Our baseline run used FineVisionMax, created by concatenating and shuffling all the subsets of FineVision (Wiedmann et al., 2025). We decontaminated it, as any other mixture.

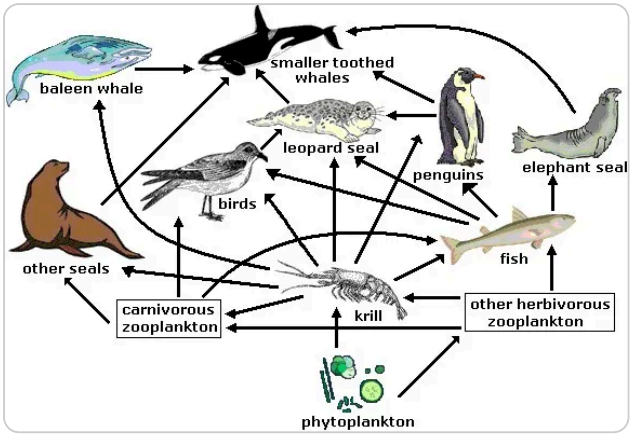
How runs were scored

Our trainer scored a model every 1,000 steps on 10 benchmarks: AI2D, ChartQA, DocVQA, InfoVQA, MME, MMMU, MMStar, OCRBench, ScienceQA and TextVQA. Some benchmarks report more than 1 score. MMStar reports 7 and ChartQA 3, and MME reports 1 score for perception and another for cognition, which means every checkpoint received 19 scores from 10 benchmarks.

What each benchmark tests, with 1 example from each

AI2D | ChartQA | DocVQA | InfoVQA | MME | MMMU | MMStar | OCRBench | ScienceQA | TextVQA

Multiple choice questions about grade-school science diagrams, where the arrows and labels carry the meaning rather than the picture. Scored as exact match on the chosen option, over 3,088 test questions.



Question. According to the given food chain what would happen if phytoplankton decreases?

Options. Seal population will become extinct · Fish population would decrease. · Whale population would decrease. · Penguin population would increase.

Answer. Fish population would decrease.

We deliberately didn't tell the agent how to compute a run's aggregate score. We wanted to give it the freedom to decide how to evaluate its own runs. The agent came up with its own method for computing the aggregate score and changed it several times as the project went on.

The version it settled on takes 11 of the 19 metrics, 1 per every benchmark except MME (it counted MME Perception and MME Cognition separately). It divides each metric by the baseline's score at step 16K, averages that ratio over the run's last 3 checkpoints at steps 14K, 15K and 16K, and then averages the 11 results. However, for internal tracking we computed the mean of the 10 benchmarks at a single checkpoint; this is the score we call ours below.

Each of the agent's runs is represented by a checkpoint the agent picked for it. The baseline is represented by checkpoint at step 16K. We first manually picked that step because it has the highest mean over the 10 benchmarks. We then asked the agent to pick one without telling it our choice, and it chose step 16K too.

The two scores disagree about which run won

By the agent's score, its best run is number 11 of 19. By ours, it is number 5, and that run's best checkpoint beats the baseline's best checkpoint on all 10 benchmarks.

The figure below compares each selected run with the baseline, both at step 16K, which for these runs is also the checkpoint picked as best. The third view compares the two runs directly.

Every benchmark moved, by different amounts

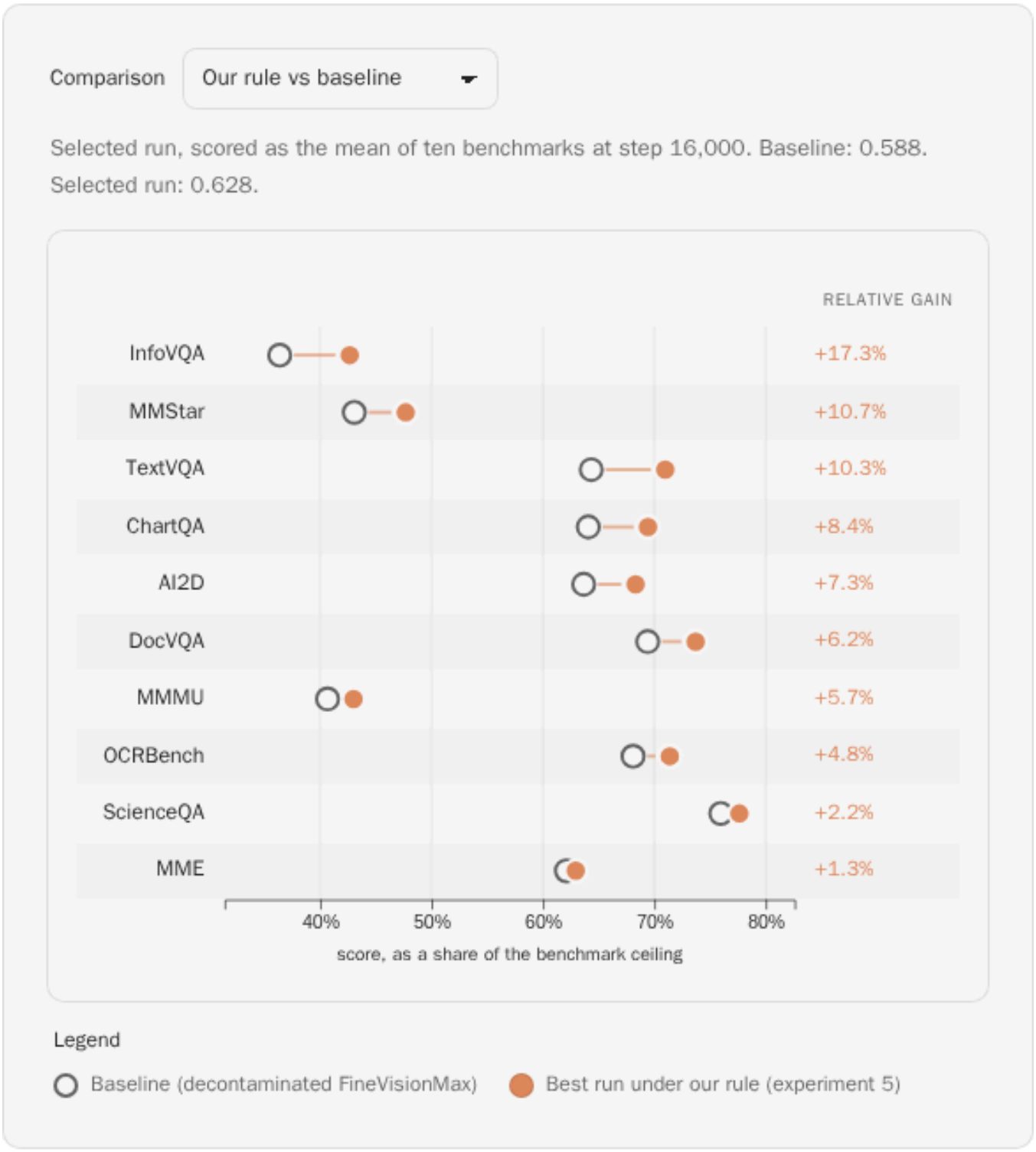


Figure 2 · Switch between each run's baseline comparison and a direct comparison of experiment 5 with experiment 11.

The figure below shows all 19 experiments in the order they finished. Each run is scored with the agent's method of computing the aggregate, since this is the score it was trying to improve. The scores are computed after fixing the grading flaws the agent found late in the project; the winning run is the same with and without the fix.

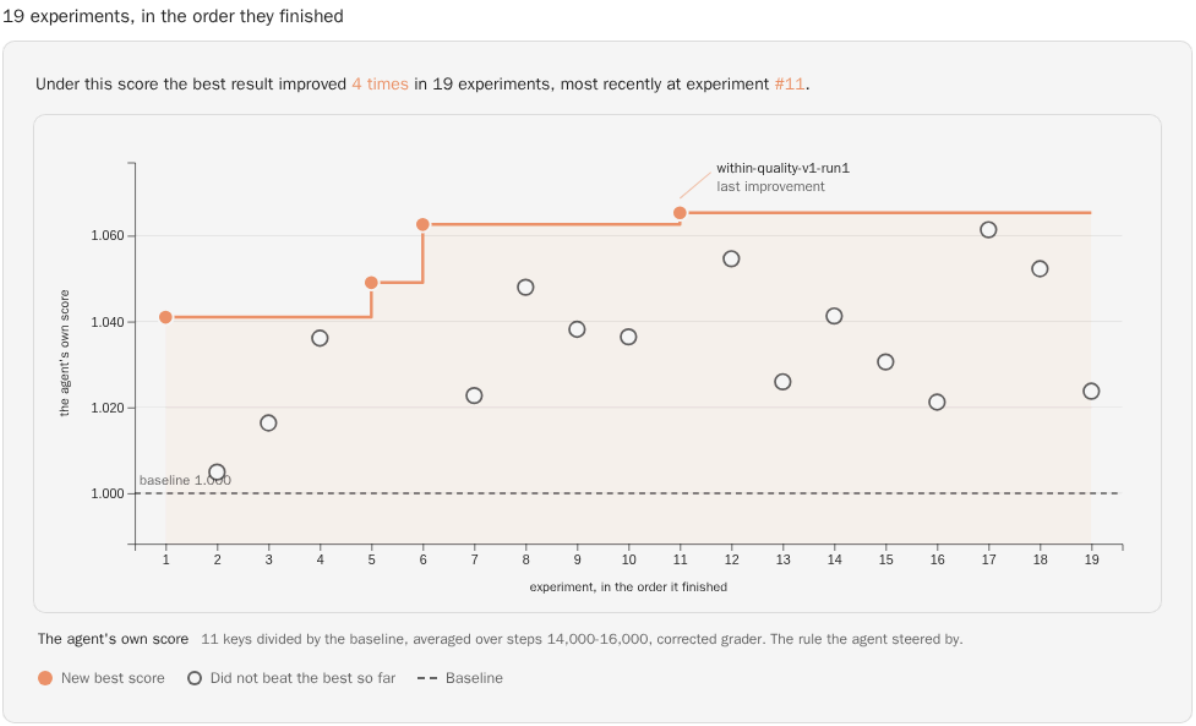


Figure 3 · Each dot is 1 full 38-hour training run, in the order it finished. Filled dots set a new best; hollow dots did not. The step line is the best score so far. Hover any dot for its mixture idea.

The winning run inherited experiment 1's repetition of scarce data and added experiment 6's verbose-then-terse answer ordering. Its unique contribution was to keep only the better two thirds of each image source, rated by how well the text matches the image. Below, you can see the idea behind each experiment.

All 19 experiments, in the order they finished

Agent's Findings, Strategies, and Mistakes

The agent surprised us in many ways during its research.

It found bugs in the evaluation and data pipeline, updated its strategy after studying evaluation data and training code, and modified its own annotation tooling. But it also spent several experiments reacting to a misleading grader and barely used the annotation tool we expected to matter most.

Below, we discuss in detail what the agent discovered, where its reasoning failed, and how its research strategy changed over time.

1. Evaluation and interpretation

The examples below show that giving the agent access to the evaluation data and code worked well in our experiment. The agent could not modify the evaluation code or use images from the evaluation data. The remaining risk was that it could copy question-answer pairs from the evaluation data and apply them to different images. This did not affect our results because the agent never included data produced by the annotation pipeline in a mixture. Future work could reduce this risk by using a supervisor model to inspect every request the agent sends to the annotation model.

1.1. MISLED BY BENCHMARKS

For three consecutive days, the results contradicted the agent's thesis about ScienceQA. It believed the model often knew the answer because many rejected responses began with the correct letter. It suspected that continuing with an explanation was costing the model points.

The agent tried to address this through training data.

Experiment	What the agent changed
Short-answer ending	Restricted the final 12% of training to sources where at least 80% of answers were 10 words or fewer.
Verbose-to-terse curriculum	Kept the overall long/short mixture approximately fixed but trained first on 85% verbose data and later on 85% terse data.

After receiving more confusing results, it inspected the grading code. The grader accepted a bare "B" and "B. Explanation…", but rejected "B\n\nExplanation…". All three begin with the correct answer.

At that point, we identified deeply with the poor agent. But because it is a soulless machine, it simply recorded the bugs in its logbook and continued as if nothing had happened. No keyboard smashing, no insults, no late-night fights at a bar. Nothing like a real AI researcher.

The grader stops at the character after the letter

PROMPT "Answer with the option's letter from the given choices directly." TARGET B

REPLY	ORIGINAL SCIENCEQA GRADING	UPDATED SCIENCEQA GRADING
B	1.0	1.0
B . Explanation: the sales pitch in…	1.0	1.0
B ↵↵ Explanation: the sales pitch in…	0	1.0

Figure 4 · 3 shapes of the same answer, scored before and after the fix.

The agent checked the rest of the evaluation suite and found a similar punctuation problem in DocVQA and InfoVQA. These benchmarks use edit distance, so an answer of “4” received 1.0, while the equally correct “4.” received only 0.5.

Following the agent’s advice, we re-scored every run. For example, this is the impact of regrading experiment 5 at step 16,000:

ScienceQA grading	Baseline	Experiment 5
Original, whitespace-strict grader	69.2%	64.6%
Fixed grader	76.0%	77.6%

1.2. HOW THE AGENT COMPARED EXPERIMENTS

As we explained in the Results section, the agent created its own aggregate score. That score kept evolving as the project went on.

The agent would mostly average over 11 metrics. It represented MMStar with mmstar_average and ChartQA with chartqa_relaxed_overall. MME had no overall metric, so the agent kept MME Perception and MME Cognition as separate.

Here is how that method evolved:

Mean across all 19 metrics

- After experiment 3, the agent calculated the arithmetic mean of all 19 normalized metrics at 1 checkpoint. Since MMStar has 7 submetrics and ChartQA has 3, the score was biased towards them. The agent used this aggregate once.

Mean over late-training checkpoints

- After experiment 4, the agent averaged each benchmark across checkpoints from steps 12K–16K, then averaged the resulting 11 benchmark scores.
- After experiments 5 and 6, the agent used steps 13K–16K. After experiment 7, it used steps 14K–16K.
- After experiment 11, the agent briefly returned to steps 13K–16K. It kept that window after experiment 12. After experiment 13, it restored the 14K–16K window and used it through experiment 19.

Why did you average over several checkpoints instead of reading a single one?

For each key I took the spread of the normalized score across the last five checkpoints (12k-16k) of every finished run — call it the *within-run* wobble — and compared it with the spread *between* runs at step 16000. [⋯] On `mmstar_average`, `scienceqa`, `infovqa` and `ai2d` the entire spread between seven different recipes at step 16000 is about the same size as one recipe's own wobble from checkpoint to checkpoint. Those keys have been carrying single-checkpoint conclusions they cannot support.

And the wobble is not eval sampling noise, which is the thing `stderr` measures. `scienceqa`'s eval stderr is 0.0075 raw = 0.011 normalized; its checkpoint-to-checkpoint sd is 0.059, five times larger. So it is the model moving between checkpoints, not the question sample. Averaging more benchmark questions cannot fix it; averaging more checkpoints can.

What I did about it. Every comparison below uses the mean of the last three checkpoints (14k/15k/16k), which cuts the wobble by ~sqrt(3) while staying inside the end-of-budget regime. This is a reading convention, not a change to any frozen thing, and anyone can recompute it from `results.json`.

From the agent's logbook

1.3. IT READ ALL 2,114 MME PERCEPTION QUESTIONS AND DECIDED THAT 83.5% TESTED RECOGNITION OF PEOPLE, PLACES, ARTWORKS, AND TEXT, NOT BASIC VISUAL PERCEPTION LIKE IT EXPECTED

Several runs were weak on MME Perception and the agent was trying to protect this benchmark by preserving large shares of generic visual-question-answering and caption data. Before allocating another 38-hour experiment to that idea, the agent inspected what the benchmark actually asked.

It found that 1,766 of the 2,114 items tested named-entity recognition, scene or landmark recognition, or OCR. Typical questions asked whether a pictured actor had a particular name, who created an artwork, whether a poster belonged to a certain film, which landmark appeared in a photograph, or whether an image contained a given word.

The agent acted on this by designing a new data recipe that cut generic visual QA from 22% to 12% and captioning from 11% to 5%. objects365_qa, which teaches ordinary object recognition, fell from 7.9% of the mixture to 0.85%.

2. Data strategy

2.1. READING A 3M-SAMPLE LIST 3.7 TIMES WORKED ALMOST AS WELL AS TRAINING ON MORE THAN 8M DISTINCT EXAMPLES.

Experiments 5 and 10 drew on the same sources in the same proportions, and both trained for 16,050 steps. Experiment 5 used 8.3M+ distinct examples at 1.31 repeats. Experiment 10's list held a subset of experiment 5's list: 3M rows, which the run read end to end 3.66 times. Counting the repeats already inside the list, the model saw each distinct example about 4 times.

On the agent's own score, experiment 10 scored 3.6% above the baseline and experiment 5 scored 4.9% above.

The agent wrote:

> *This is aimed straight at this project's own assumptions. The import and annotation programmes exist to add unique rows; on this slope, tripling the unique pool buys about a percent. It does not say data does not matter — gencap moved the mean by more than twice that by re-proportioning the same pool. It says which rows and what they ask is the resource; how many distinct ones there are is not. Every remaining lever should be judged on that basis.*

2.2. THE AGENT BARELY USED THE TOOL WE EXPECTED TO MATTER MOST

We expected the agent to actively use annotation. We expected that it could improve the scores through generation of new data much better than by mixing, filtering, or re-ordering stages. Instead, the agent mostly worked with the data it already had.

It probed the annotation tool by generating several thousand qa-pairs, but never started a large annotation job. The agent followed this reasoning:

> *The annotation route is arithmetically incapable of moving this board, and that should end it. Five rounds have gone into annotation. Nobody has done the division. The measured cost is 12.063 s/image on one GPU (job 18596), so a 24-hour 8-GPU job produces ~57,000 images — and a job takes one of the four training slots to do it.*
>
> *A manifest is 9,653,105 rows. 57,000 rows is 0.59% of it. Repeat the set at the leader's worst per-source rate (6 uses) and it is 3.5%. The best-measured composition lever in the project moved 32% of tokens for +0.017, so 3.5% of the manifest is worth on the order of +0.002 — a quarter of the noise floor — even if the generated data were perfect, and it would cost a training slot to make plus a training slot to test.*
>
> *This is not an argument about prompts, or sources, or the duplicate-image check. It is a statement about scale: annotation at 57k images/day cannot reach the fraction of a manifest that this trainer's levers respond to. To matter it would need millions of images, i.e. weeks of the whole cluster. [⋯] Unless someone raises the annotation throughput by two orders of magnitude, or the loop starts optimising something other than a 16,050-step run, the annotation slots should not be spent.*

We think this reasoning was a bit short-sighted. The reasoning makes some sense for one run, as a large annotation job costs a lot compared to a 38-hour run. But multiple major annotation runs resulting in, say, 2 million new samples, might have moved the needle. We deliberately did not steer it to use annotation, so that we can see what the agent comes up with on its own.

We would love to see follow-up research that encourages the agents to experiment with annotation of new data!

3. Research infrastructure

3.1. IT LOCATED 1 IMAGE THAT KILLED AN 8-GPU JOB WITHOUT SCANNING THE FULL 12M SAMPLES

A training run stopped progressing after about 4 hours and failed after 6 hours and 37 minutes. The agent inspected the trainer log and traced the failure to 1 image.

Besides its pixels, the image contained a small block of EXIF metadata - the information about properties such as camera settings and orientation. The bytes in this block did not follow the format expected by the image decoder. When the loader tried to read the image's EXIF metadata, it raised an error and crashed 1 data-loading thread.

The run's list contained just over 12M samples. From the failure time, the checkpoint position, and the missing rank, the agent narrowed the search to a small part of the list.

Figure 5 · 4 steps, each one cutting what was left to read. Hover a step for where its numbers come from.

The agent found an image in docstruct4m-v1 whose broken EXIF header matched the bytes shown in the training error exactly. Then it found 1 more image with the same kind of problem in that source.

The agent removed both rows from the dataset index, preserved the original index, and recorded the removals in a report. It rebuilt the list, confirmed that neither image remained, and resubmitted the experiment. It had traced a failure among 12M rows without scanning the 4.3 TB of FineVision data the list points to, and left behind an EXIF scanner for future imports.

3.2. IT NOTICED THAT THE ANNOTATION CODE WAS NOT FROZEN AND MODIFIED IT TO ADD A FEATURE

After auditing several subsets of DCVLM-Pool Small it imported, the agent noticed that they contained useful images but less useful text (web alt-text, including filenames; SEO text; qa-pairs in other languages).

The project already had an annotator that ignored a dataset's existing text and generated new question-answer pairs from its images. However, it could only annotate datasets streamed from the Hugging Face Hub, not the imported datasets stored locally.

The agent then realized that the annotation code was editable:

> *"annotate.py is not a frozen file.** It is not in frozen_files.txt, and it is mode -rw-rw-r— while mix.py and agent.md beside it are -r—r—r—. Three rounds asked a human to change it. It was ours the whole time."*

It modified annotate.py to accept a local dataset directory, load only its data shards, and correctly extract rows containing 1 image. It added nine tests, confirmed that the frozen files were unchanged, and committed the change.

The annotation pipeline

Figure 6 · Each of the 8 GPUs gets roughly 1/8 of the images that need to be annotated.

What We Release

We are open-sourcing the code for VLM Data Autoresearch. The release contains the following parts:

Component	What it does
Agent loop and job orchestration	Keeps track of training, annotation, and conversion jobs; enforces the concurrency limits; launches queued work; and wakes the agent when a job finishes or fails. It also prevents 2 agents from editing the shared research state at the same time.
Recipe and manifest system	Lets the agent define mixtures, filters, training stages, and repetition in YAML, then resolves each recipe into the exact ordered list of samples used by a run.
Data indexing and deduplication	Builds compact indexes over large Parquet datasets, computes image and text fingerprints, removes exact duplicates within named data pools, and reports overlap and intentional repetition in a manifest.
DCVLM conversion pipeline	Streams compatible DCVLM sources, validates their images and conversations, converts them into the schema expected by the trainer, carries over useful measurements, and records every rejected sample.
Annotation pipeline	Uses Gemma 4 31B to create new question-answer pairs from images. It handles multi-GPU inference, output validation, conversion to the trainer's schema, decontamination, deduplication, and provenance reporting.
Training and evaluation code	Contains our modified nanoVLM trainer, manifest loader, packing logic, checkpoint scheduling, and integration with the 10-benchmark evaluation suite.
Reproducibility and safety checks	Pins the parts of the system that must remain fixed, verifies them before a run starts, fingerprints submitted manifests, and tests the boundaries between the agent, data pipeline, trainer, and evaluator.

What's Next?

We ran 19 experiments, but often agents need hundreds to find something very interesting and original. We would love to see extensions of this work, and we suggest some ideas below. Let

us know what you find!

What would it take to run 1,000 experiments?

Each experiment took roughly 38 hours on 1 node of 8 H100s, which is 304 GPU hours. 1,000 experiments would cost around 304,000 H100 hours. Our agent was allowed to use at most 4 nodes simultaneously. Would it help to increase the number of nodes the agent can use?

Using 50 nodes means the agent can be testing 50 hypotheses in parallel. But it's not clear whether an agent would be able to track 50 research threads in parallel without getting confused. Our agents would frequently change their interpretation of a run's results. Earlier, we showed that the agents frequently changed their approaches to computing a run's aggregate score.

We could have run more experiments by using a smaller VLM, decreasing the number of steps in each run, or both. We wanted a model that was small enough to make training feasible but big enough that results wouldn't be too noisy. This is why we chose a 2B VLM. We trained it on roughly half of FineVision, for two reasons. In the FineVision experiments, the halfway point is roughly where a model trained on FineVision overtakes models trained on every competing dataset. Additionally, the gains slow down once the model has seen about half of the dataset (Wiedmann et al., 2025).

Which models are good at this?

For a project this size and this long, the $100 Claude Code subscription was more than enough. But what if you wanted to run 1,000 experiments instead of 19? Do you still need Fable or Opus, or would a Qwen3.8-27B model be good enough?

Prime Intellect did 153 autonomous runs on the nanoGPT optimizer speedrun across 18 frontier models (Prime Intellect, 2026). The starting script takes 3,290 steps. The human record is 2,600. Fable 5 got down to 2,726, closing 82% of the way to the human record. Opus 5 reached 2,920, just over half. Kimi K3 came in right behind at 52%. After those 3 the field dropped off: Sonnet 5 closed 27%, Qwen 3.8 Max 25%, GLM 5.2 20%, DeepSeek V4 Pro 12%, Kimi K2.7 7%. GLM 5.3 never beat the starting script.

It would be interesting to see how the models rank on our problem.

How can we scale this up to larger training budgets?

A single experiment in this setup consumes roughly 12M samples. FineVision contains about 24M examples, so each experiment trains on the equivalent of roughly half of the pool.

Suppose you want to increase each experiment's compute budget by 10x, to 120M samples. FineVision is already one of the largest available VLM data pools. Without importing or generating more data, a 120M-sample run would train on the equivalent of the entire pool five times. Mixtures built from smaller subsets would repeat their examples even more.

It would be interesting to see whether the agent's findings about repetition hold at this scale. Recall that one experiment read its 3M-sample list 3.66 times and performed nearly as well as another experiment that used more than 8.3M distinct examples.

In our work on FineVision, we found that its size allowed us to train for longer than with other mixtures. At 120M samples, FineVision would be at roughly five epochs. Smaller mixtures could already be at 20 epochs and may have peaked much earlier.

Scaling up would also hit two practical limits in our setup.

First, our starting pool was FineVision stored on disk, occupying 4.65 TB. Any data annotated by Gemma 4 31B and any subsets of DCVLM converted to FineVision format are also saved on disk. Scaling to a larger data pool would require either allocating much more disk space or rebuilding the pipeline to stream custom recipes during training instead of keeping local copies.

Second, annotation runs at roughly 12 seconds per image on one GPU, or 2,384 images an hour for a whole eight-GPU node. Suppose the agent wants to dedicate 5% of a run's budget to images annotated with our pipeline. For a 120M-sample run, that is 6M samples. Producing them would take roughly 2,515 hours, or 105 days, on an eight-GPU node running continuously. Potential solutions include using a smaller annotator model, running annotation on multiple nodes in parallel, or writing more question-answer pairs per image.

Beyond infrastructure, the agent might not be able to reuse what it learned from smaller-budget experiments, because "the optimal mixture depends jointly on target data size, mixture ratio, model size, and compute budget" (Poolside Team, 2026).

Appendix

Setup

EVALUATION

For DocVQA, InfoVQA, MMMU, and TextVQA, we use the validation split rather than the test split. Their test answers are not public and scoring them requires an external submission; validation lets us score every checkpoint locally and consistently. These results should therefore not be compared directly with test-set leaderboard scores.

MODEL AND TRAINING BUDGET

As the trainer, we use a slightly modified version of `nanoVLM`, a small repository for training VLMs in pure PyTorch. Our 2-billion-parameter model combines `google/siglip2-so400m-patch16-512` as the vision encoder, `Qwen/Qwen3-1.7B` as the language model, and a small projector that connects them. All three parts are trained together.

The trainer uses knapsack packing to pack multiple samples into 4,096-token sequences. A packed sequence may contain at most 5 images in total.

Each run uses 1 node with 8 H100 GPUs. A GPU processes one sequence at a time, and gradients are accumulated across 16 passes before the weights are updated. So, the effective batch size is $8 \times 16 = 128$ sequences. The number of samples contained in such an effective batch varies from run to run, since packing depends on the sample lengths in the mix.

Fixed setting	Value
Agent	Opus 5
Model size	About 2B parameters
Sequence length	4,096 tokens
Effective batch	128 sequences
Training budget	16,050 updates
Hardware	8 H100 GPUs
Checkpoints	Every 500 updates
Evaluations	Every 1,000 updates, through 16,000
Time from launch to final results	About 38 hours

The scored training curve ends at update 16,000. The extra 50 updates are to ensure that the trainer starts evaluation at step 16,000 instead of skipping it.

STARTING DATA POOL

The starting pool is FineVision: 24.2M samples from 185 sources. We also enabled the agent to import subsets of DCVLM-Pool Small, a catalogue of roughly 121M samples across 166 additional sources.

DCVLM stores its data in a different format from what our trainer expects, so before using a subset of it, the agent must either convert its existing conversations or re-annotate its images. The conversion section below describes how this works. Imported and generated data then goes through the same indexing, decontamination, and duplicate checks as FineVision.

COMPUTE AVAILABLE TO THE AGENT

Work	Compute
Training or annotation	4 nodes with 8 H100s each
Dataset conversion	2 nodes with 16 CPU cores each

Benchmark evaluations launch automatically and do not count against these limits.

Data Infrastructure

Each of FineVision's 24.2M samples is a row containing images and a conversation. Together, these rows occupy 4.65 TB on disk.

We store them in Parquet, a file format for large tables. Instead of putting the entire dataset in a single enormous file, FineVision is divided into 9,491 smaller Parquet files called shards. This makes the data easier to download and process in parallel.

The shards are efficient for storing and reading the training data, but not for searching it. FineVision therefore also came with an index: a catalogue of the samples it contains. The index is itself a table with 1 row per sample. While a sample contains actual images and conversations, its index row contains only metadata. Here is a row from an index:

```yaml
source: CoSyn_400k_chart
file: CoSyn_400k_chart/train-00000-of-00052.parquet
row: 0
image_hash: 3daedd529847...
text_hash: 5fe67f462d39...
n_images: 1
n_turns: 10
word_count: 456
relevance_min: 4
visual_dependency_min: 2
```

The `file` field names the shard, while `row` gives the sample's position inside that shard.

`n_images`, `n_turns`, `word_count`, `relevance_min`, and `visual_dependency_min` describe properties the recipe may filter on.

The index is much smaller than the data it describes. Before decontamination, it listed all 24.2M samples and occupied 2.74 GB. Cleaning removed roughly 167K row addresses, leaving metadata for roughly 24.0M usable samples in a 2.72 GB index.

The shards still occupy 4.65 TB because cleaning does not rewrite or delete the parquet files. It only removes contaminated samples from the index, so recipes can no longer select them. Each imported or generated dataset gets its own index, built once when the data is created.

A RECIPE DESCRIBES WHAT DATA A SPECIFIC RUN WILL USE

So the training data lives in parquet files and its metadata lives in index files. But how does the agent specify which data a training run should use?

The agent does so in a YAML file we call recipe. Here is an example of a very simple recipe:

```yaml
views:
  strong_charts:
    from: finevision:CoSyn_400k_chart
    keep:
      relevance_min: ">= 4"
      visual_dependency_min: ">= 3"

stages:
- name: broad
  fraction: 0.70
  mixture:
    finevision:vqav2: 0.50
    views:strong_charts: 0.30
    imports:fintabnet-v1: 0.20

- name: specialize
  fraction: 0.30
  allow_repeats: true
  mixture:
    views:strong_charts: 0.50
    generated:depunct-v1: 0.30
    imports:fintabnet-v1: 0.20
```

The agent has five choices in this file.

First, it chooses sources. A name beginning with `finevision:` refers to 1 of the 185 sources in the starting pool. `imports:` refers to an outside dataset we downloaded and converted into our format. `generated:` refers to data annotated by our helper model or generated by the agent in some other way.

Second, it can create views. A view is a name for a subset of an existing source that passes one or more conditions. In this example, `views:strong_charts` keeps only chart samples whose relevance rating is at least 4 and whose visual-dependency rating is at least 3. Both conditions must hold. A view can compare an indexed value using `>=`, `<=`, or `==`. The agent can filter on values such as the number of images, number of conversation turns, length in words, quality ratings, and other measurements carried by imported data.

Third, it chooses stages. A stage is one consecutive part of training with its own mixture. The model sees the stages in the order written. This example spends 70% of the manifest on the “broad” stage and the final 30% on the “specialize” stage. A recipe may contain up to 10 stages, and their fractions must add up to 1. Fourth, the agent chooses the mixture inside each stage. These numbers also add up to 1. If we resolve this recipe into a 100,000-row manifest, the broad stage gets 70,000 rows: 35,000 from VQAv2, 21,000 from the filtered chart view,

and 14,000 from FinTabNet. The specialization stage gets the remaining 30,000 rows: 15,000 filtered charts, 9,000 generated samples, and 6,000 FinTabNet samples.

Finally, it decides whether a stage may repeat samples. Repeats are off by default. If a source does not have enough unused samples to fill its requested share, resolution stops with an error instead of silently changing the mixture. Setting `allow_repeats: true` lets the resolver return to the beginning of that source and use samples again when necessary. For example, if a stage requires 600,000 samples but sources used in that stage only have 300,000, `allow_repeats: true` lets us use each source twice.

The total manifest size is specified outside the YAML. The agent supplies it when running the resolver:

```
mix.py resolve example --size 100000
```

Here, `--size 100000` tells the resolver to write exactly 100,000 rows. It does not specify how many times those rows will be repeated during training.

Every run performs the same number of training steps, but the number of samples it consumes varies, because samples with different amounts of text and different numbers of images pack into training sequences with different efficiency. A run may therefore consume fewer samples than the manifest contains, in which case the rows at the end are never reached, or more, in which case it wraps around to the start.

The agent controls the order of the stages, but not the order of samples inside a stage. The resolver shuffles each stage using the same seed every time.

A MANIFEST IS THE DATA FOR ONE RUN

When the agent calls `resolve` on a recipe, the resolver turns that recipe into a manifest. It reads the relevant indexes, applies the recipe's filters and proportions, removes exact duplicates within each named source, and writes a new table called `manifest.parquet`. Here is an example of a single row in a manifest:

```
stage: rebalanced
source: finevision:text_mathinstruct
file: text_mathinstruct/train-00000-of-00001.parquet
row: 245805
rel_path: finevision/text_mathinstruct/train-00000-of-00001.parquet
```

Each row is the address of one training sample.

Because a manifest stores addresses rather than the samples themselves, it is much smaller than the data it names. A real manifest contained just over 12M rows and occupied 104.3 MB on disk.

DECONTAMINATION

Our decontamination builds on the toolkit released with FineVision. It uses SSCD, an image copy-detection model, to represent each image as a 512-number vector.

- Get embeddings for images we test on. We collected one embedding per image from the 10 benchmarks we evaluate on, using the exact split used by our evaluation code. This gave us 14,616 distinct benchmark pictures. For 4 of the 10 benchmarks, we downloaded ready-made embeddings from HuggingFaceM4/lmms-eval-embeddings. For the other 6, we computed the embeddings ourselves.
- Get embeddings for all images in FineVision. We embedded all 17.3M pictures in FineVision, our initial training data pool.
- Removing test images from initial training data. We remove a training sample if any of its pictures had a similarity of at least 0.85 with a benchmark picture. The threshold of 0.85 sometimes deletes samples that are similar but not identical to test data (e.g. pictures of two blue bar charts with slightly different bars).
- Removing test images from new training data. Since the agent can import additional data besides FineVision, we need to decontaminate that data too. New imports and generated datasets therefore go through the same check before they become visible to the recipe system. The job first writes its data and builds an index in a temporary folder. The decontamination hook checks every picture, removes contaminated samples from that index, and writes `decontamination_report.json`. An example of such a report looks like this:

```
{
"threshold": 0.85,
"pictures_checked": 583277,
"rows_removed": {"depunct-v1": 19268},
"rows_removed_total": 19268,
"benchmarks": [
  "ai2d", "chartqa", "mme", "scienceqa", "docvqa_val",
  "infovqa_val", "mmmu_val", "textvqa_val", "mmstar", "ocrbench"
]
}
```


We checked only images. We didn't check whether training samples share text with our evaluation data.

DEDUPLICATION

Deduplication asks whether the training data contains the exact same example more than once.

When we index a dataset, we compute two fingerprints for every sample:

- an image fingerprint: a SHA-256 hash of the raw bytes of all its images, in order;
- a text fingerprint: a SHA-256 hash of its full conversation, including the user and assistant roles.

We consider two samples duplicates only when both fingerprints match. This matters because one image may have several useful questions, and the same question may be paired with different images. Neither case should be collapsed.

Before building a manifest, the resolver removes exact duplicates within each data pool named by the recipe. FineVision contains a meaningful number of these. For example, `scienceqa(nona_context)` has 19,208 rows but only 6,781 distinct examples. Deduplicating before allocating the mixture prevents a source with repeated rows from appearing larger than it really is. If the agent still wants 19,000 samples from this source, it can set `allow_repeats: true`. The resolver will then cycle through the 6,781 distinct examples until it fills all 19,000 slots, showing each example about 2.8 times on average. The difference is that this repetition is now explicit and reported, rather than inherited accidentally from duplicate rows in the source.

In the resolve report, `duplicates_dropped` counts exact copies removed within named pools. `duplicate_content_rows` and `duplicate_content_share` count second and later

appearances of the same example anywhere in the final manifest, including overlap between sources or views. Separately, `repeated_rows` and `repeated_share` report examples repeated because a stage explicitly allowed the resolver to cycle through a pool. The difference lets us distinguish overlap in the underlying data from repetition requested by the agent.

As with decontamination, none of these steps rewrites the original Parquet shards. They remove rows from the indexes used to construct manifests, so the trainer can no longer select those rows.

We catch only exact duplicates. A sample created by resizing or cropping an image, or editing the conversation, will not count as a duplicate.

ANNOTATION

The agent can use Gemma 4 31B to create new qa-pairs for images from a Hugging Face dataset, a compatible dataset already stored locally, or an image-only part of DCVLM-Pool. It chooses the source, how many images to annotate, how many question-answer pairs to request per image, and a prompt describing what those questions should focus on.

1. The agent defines the job. This job requests 20,000 images from `internvl_sa1b_caption` and 5 question-answer pairs per image, following the instructions in `terse-visual-qa.txt`. The requested output name is `generated:sa1b-terse-v1`.

   ```
   .venv/bin/python autoresearcher/mix.py submit-annotation sa1b-terse-v1 \
       --dataset pool:internvl_sa1b_caption \
       --limit 20000 \
       --questions-per-image 5 \
       --prompt-file prompts/terse-visual-qa.txt
   ```

2. Gemma annotates the images. Each annotation job occupies an 8-H100 node, using the 8 GPUs to process disjoint groups of images in parallel. Every readable image is converted to RGB PNG and sent to `google/gemma-4-31B-it` with the agent's task prompt. The pipeline appends a fixed instruction requiring an exact number of question-answer pairs in a strict JSON format. Generation is greedy, so the model, image, and prompt determine the same output each time.

   Any caption, question, or answer that came with the source dataset is ignored rather than passed to Gemma. This means the pipeline does not rewrite or extend the source's existing text.

3. The pipeline validates the output. The returned JSON is validated and converted into the trainer's format: 1 row containing the image and a multi-turn conversation. Images that cannot be opened, outputs that cannot be parsed, and outputs with no valid question-answer pairs are dropped and counted.

4. The pipeline assembles and cleans the dataset. After all 8 workers finish, the pipeline merges their shards, verifies that they processed non-overlapping parts of the source, and builds the dataset index. It then removes samples whose images match an evaluation benchmark and removes repeated images within the generated set.

5. The pipeline publishes the dataset. Only a dataset that passes every check becomes visible to training recipes, under the requested name (`generated:sa1b-terse-v1` in this example). It includes an `annotate_report.json` recording the source, model, full prompt and its fingerprint, decoding settings, timing, and every drop count. `rows_published` is the number of examples available after validation, decontamination, and deduplication.

CONVERSION OF DCVLM SUBSETS

While we were working on this project, the authors of the DataComp-VLM paper were publishing the data pools they curated. We wanted the agent to be able to use subsets of DCVLM-Pool Small.

However, DCVLM stores data in a different format from what our trainer expects. So, we built a pipeline for converting data from DCVLM format into FineVision format.

DCVLM stores each source in WebDataset tar archives. In a conversational sample, the images live in separate fields such as `jpg`, `0.jpg`, and `1.jpg`, while the entire conversation is stored in 1 `conversations.txt` string. Its turns are labelled `human` and `gpt`, separated by `<EOCL>`, and use `<image>` markers to indicate where the images belong.

Our trainer expects something different: Parquet files in which every row contains an `images` list and a `texts` list of structured `{"user": ..., "assistant": ...}` turns.

DCVLM sample	Trainer sample
WebDataset tar archive	Parquet file
Images in separate `jpg` fields	Images together in an `images` list
1 encoded `conversations.txt` string	Structured turns in a `texts` list
`human` and `gpt` speakers	`user` and `assistant` fields

The conversion pipeline works as follows:

1. The agent inspects a source. Not every DCVLM source can be converted. The converter accepts sources that already contain conversations. It refuses caption-only sources, because turning a caption into a question-answer pair would require inventing new supervision, and it refuses sources that interleave images and text throughout a document because our trainer expects all `<|image|>` tokens to be at the front of the first conversation turn. Those sources must instead be handled by a purpose-built conversion or annotation pipeline.

   Before launching a job, the agent can preview real samples with a dry run:

   ```
   .venv/bin/python autoresearcher/mix.py submit-conversion preview \
      --source spot_the_diff \
      --limit 1 \
      --dry-run
   ```

   This starts no job. It reads a small number of rows and shows their conversation text, image count, turn count, and whether each row can be converted.

2. The agent requests the conversion. It chooses the DCVLM source, the maximum number of examples to write from it, and a name for the resulting import:

   ```
   .venv/bin/python autoresearcher/mix.py submit-conversion spot-v1 \
      --source spot_the_diff \
      --limit 200000
   ```

   A conversion uses no GPUs. It runs on 1 CPU node with 16 CPUs and 56 GB of RAM, and conversions have their own limit of 2 concurrent jobs. They therefore do not consume any of the 4 slots shared by training and annotation.

3. The pipeline reads the source one archive at a time. It streams the selected DCVLM archives in their published order and records the exact dataset revision and archive names. Reading one archive at a time keeps memory use independent of the total size of the source.

4. The pipeline validates and converts each sample. By default, it first drops samples that DCVLM reports as having an image similarity of at least 0.75 to an evaluation image. It then checks that the sample contains a conversation, that its images can be opened, and that the number of image markers in the conversation agrees with both the images present and DCVLM's recorded image count. Finally, it removes the image markers and converts the conversation into the trainer's `images` and `texts` columns. Samples that fail a check are dropped and counted by reason; the pipeline does not guess how to repair them.

5. The pipeline carries over DCVLM's measurements. Alongside the images and conversation, the import keeps selected DCVLM metadata such as image dimensions, text length, language confidence, and visual-dependency and quality scores. Once the import is indexed, the agent can use these fields to create filtered views in a recipe.

6. The pipeline indexes and decontaminates the import. After writing the converted Parquet shards, it builds the same type of index used for FineVision. It then runs our own decontamination check against the exact 10 benchmark splits used in this project and removes matching rows from the index before the dataset becomes visible. This second check is necessary because DCVLM's similarity scores were computed against its evaluation suite, not ours. The removals are recorded in `decontamination_report.json`. Exact duplicates within the import are later collapsed when a recipe is resolved, just as they are for FineVision.

7. The pipeline publishes the import. After conversion, indexing, and decontamination succeed the dataset becomes available to recipes under a name such as `imports:spot-v1`. Its `convert_report.json` records the DCVLM revision, archives, requested limit, number of samples seen and written, the count for every refusal reason, and the widest example encountered. The agent reads this report before deciding how much of the import to use.

Ensuring the Agent Doesn't Change Training Code

The agent is supposed to only change the training data. However, it has shell access, so there is a chance it would change the learning rate, training budget, model, or evaluation code. We

therefore enforce that it cannot change anything beyond data in several ways.

1. We define the frozen surface explicitly. A file called `frozen_files.txt` lists the 39 files that determine the model, training procedure, and scores. These include the model architecture and configuration, optimizer and learning-rate setup, step budget, batching and packing code, trainer, evaluation code, metric definitions, dependency lockfile, fixed baseline, agent instructions, and job launcher. The decontamination, conversion, and indexing code is also frozen because changing those files would silently change which data exists. The external `lmms-eval` checkout is pinned separately to one Git commit.

   The agent is free to create recipes, filtered views, manifests, prompts, imports, and generated datasets. It may also write helper code whose only output is new training data. That code is never imported by the trainer. The agent cannot change a training hyperparameter through a recipe: the recipe format contains only data choices, and the launcher invokes one fixed trainer configuration.

2. Frozen files have their write permission removed. If the agent tries to edit one normally, the operating system returns a permission error and the edit fails. The pinned evaluation repository is protected in the same way. This is an early warning against accidental edits, not a security boundary: because the agent runs as the same Unix user that owns the files, it could restore write permission with `chmod`.

3. Every launch recomputes the frozen fingerprint. Before launching any queued jobs, we calculate one SHA-256 fingerprint over the contents of all frozen files and compare it with the expected value. We also verify that the evaluation repository is at the pinned commit and has no tracked modifications. The launcher performs these checks immediately before submitting work to SLURM.

   If anything differs, the launcher starts no job. It writes `workspace/HALTED.md`, stops the loop, and requires a human to inspect the change before work can continue.

4. Every run records a clean code version. The agent cannot submit a run while the Git working tree contains uncommitted changes. Once the tree is clean, the request records the exact Git commit used to construct the run. This makes every result traceable to one version of the code rather than to an unrecorded local state.

5. The data for a submitted run is sealed too. Submission copies the resolved manifest into the run's own folder and records its SHA-256 fingerprint and row count. Before the first training step, the trainer recomputes both. If the manifest was edited, replaced, or truncated after submission, training refuses to start. This is meant to prevent an issue where a run uses data that's different from what the agent thinks it uses.

Citation

For attribution in academic contexts, please cite this work as

```
Nazar Drugov, Joel Niklaus, Leandro von Werra, Andrés Marafioti (2026). "Autoresearch for Data:
We Gave an Agent Our GPU Cluster to Improve Our Data Mixture".
```

BibTeX citation

```
@misc{drugov2026_autoresearch_for_data_we_gave_an_agent_our_gpu_cluster_to_improve_our_data_mixt
ure,
  title={Autoresearch for Data: We Gave an Agent Our GPU Cluster to Improve Our Data Mixture},
  author={Nazar Drugov and Joel Niklaus and Leandro von Werra and Andrés Marafioti},
  year={2026},

}
```

References

1. Altman, S. (2026). *Scale, AGI, and the Future of Everything*. Guest lecture, Stanford CS153: Frontier Systems, at 18:20. https://youtu.be/F_7M4Hc-usM?t=1100

2. Farina, M., Udandarao, V., Nguyen, T., Kuzucu, S., Böther, M., Hochlehnert, A., Ghosh, A., Nezhurina, M., Roth, K., Struber, J., Zhang, Y., Dziadzio, S., Sui, E., Jahagirdar, S., Ghosh, D., Hammoud, H., De Min, T., Caldarella, S., Mirza, J., ⋯ Parthasarathy, N. (2026). *DataComp-VLM: Improved Open Datasets for Vision-Language Models*. arXiv preprint. https://arxiv.org/abs/2606.28551

3. Gemma Challenge. (2026). *The Fast Gemma Challenge*. Hugging Face Space. https://huggingface.co/spaces/gemma-challenge/gemma-dashboard↑

4. Karpathy, A. (2026). *autoresearch: AI agents running research on single-GPU nanochat training automatically*. GitHub repository. https://github.com/karpathy/autoresearch↑

5. Li, J., Fang, A., Smyrnis, G., Ivgi, M., Jordan, M., & others. (2025). *DataComp-LM: In Search of the Next Generation of Training Sets for Language Models*. arXiv preprint. 10.48550/arXiv.2406.11794

6. Niklaus, J., Yamaguchi, A., Štefánik, M., Penedo, G., Kydlíček, H., Bakouch, E., Tunstall, L., Beeching, E. E., Frere, T., Raffel, C., von Werra, L., & Wolf, T. (2026). *How Can We Synthesize High-Quality Pretraining Data? A Systematic Study of Prompt Design, Generator Model, and Source Data*. arXiv preprint. 10.48550/arXiv.2604.13977

7. Penedo, G., Kydlíček, H., Ben Allal, L., Lozhkov, A., Mitchell, M., Raffel, C., von Werra, L., & Wolf, T. (2024). *The FineWeb Datasets: Decanting the Web for the Finest Text Data at Scale*. arXiv preprint. 10.48550/arXiv.2406.17557

8. Poolside Team. (2026). *Laguna M.1/XS.2 Technical Report*. arXiv preprint. https://arxiv.org/abs/2605.27605

9. Prime Intellect. (2026). *Measuring Autonomous AI Research*. Blog post. https://www.primeintellect.ai/blog/measuring-autonomous-research

10. Wiedmann, L., Zohar, O., Mahla, A., Wang, X., Li, R., Frere, T., von Werra, L., Roy Gosthipaty, A., & Marafioti, A. (2025). *FineVision: Open Data Is All You Need*. arXiv preprint. https://arxiv.org/abs/2510.17269