IMPROVING VLMS ON SCIENTIFIC CHART ANALYSIS VIA AUTONOMOUS EVOLUTION

Anonymous authors
Paper under double-blind review

ABSTRACT

Understanding and operating with scientific charts is a standard approach in data analysis, and while humans are good at interpreting such charts, open vision-language models (VLMs) are known to perform poorly when it comes to question answering based on such data types. Motivated by an initial observation that many of their errors are due to inexact answering, we raise the following natural question: *how much accuracy can we recover without changing the model at all?* In this paper, we study this question by searching over inference programs and the code around a "frozen" version of the model. Using ideas from autonomous evolution (famously used in AlphaEvolve), we first start from a baseline inference program, and in each round we show an LLM-judge the code together with a few failures on training questions, and ask for an improving evolutionary move. As usual in autonomous evolution approaches, the candidate programs are kept in a small pool ranked by training accuracy, and the next edit is performed on top-scorers from this pool. We run this approach on three VLMs and four popular benchmarks on scientific charts, and we measure the fraction of descriptive questions whose answer exactly matches the ground truth. The accuracy improves in all experimental settings, and on average, it rises from 16.6% with the starting program to 41.3% with the evolved one, and the largest single increase is 55.2 percentage points. As a side benefit, our evolved program is 3.8× faster than the starting program, because short answers are easier to grade and faster to generate. Our results demonstrate that a large part of the observed errors arise due to the inference pipeline, and that optimizing it can improve the "frozen" VLM without the need to retrain it.

1 INTRODUCTION

Plots are one of the key mediums by which scientific results are often reported, and analyzing them is an increasingly common task that is being given as input to vision-language models (VLMs). Unfortunately, while humans are good at interpreting and answering questions based on such plots, VLMs often struggle (Masry et al., 2022; Yue et al., 2024).

Just to give some examples of these requests, they may include tasks such as *"report the highest tick on an axis"*, or *"name the series with the largest gap"*, or *"extract the value of a bar"* and more. It turns out that open-weight VLMs remain far below human accuracy on benchmarks built from real scientific figures (Masry et al., 2022). Here, we use a variety of datasets: CharXiv, ChartQA, PlotQA, and EvoChartQA. CharXiv (Wang et al., 2024) is a popular dataset curated by charts drawn from arXiv papers. It is reported that even strong models have a relatively low rate of correctness for answering descriptive questions compared to humans, and the gap widens on reasoning questions. ChartQA (Masry et al., 2022), PlotQA (Methani et al., 2020), and EvoChartQA (Huang et al., 2025) demonstrate a similar phenomenon across real-world, synthetic, and long-tail charts, with the gap being relatively largest for the small open models that are cheapest and easiest to deploy.

This work started from the failure modes of VLMs on such input data, and observing *how* these models fail is quite instructive. For example, when asked for *"the highest labeled y-tick on a chart"*, Qwen3-VL-2B (Bai et al., 2025) replied *"The y-axis label is ... the highest tick value is 58 ..."*, even though the ground truth was **60**. Upon being asked for a percentage, the model responds, *"The value is 28%"*; however, the ground truth was **38**. As observed, the model fails to read the chart

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

correctly in both cases, and since descriptive questions are graded by exact match, both answers are classified as wrong.

More generally, inspecting the outputs of the baseline inference pipeline (Table 3) shows that a large share of the "errors" are of similar flavor, i.e., the VLMs either provide incorrect values, or sometimes correct values may be present but wrapped in a heavily verbose sentence, followed by a unit or percent sign (or even preceded by the model's intermediate arithmetic). Perhaps surprisingly, even with a plain prompt and a generous token budget, Qwen3-VL-2B scores only 0.5% on ChartQA descriptive questions and 0.0% on PlotQA! These are observed to be failures of presentation and also perception.

The above observations raise the main driving question behind our work:

Main question: *How much accuracy in scientific chart understanding can we recover, without changing the model at all?*

Baseline Inference Program. Everything that lies between a benchmark question and the evaluated string is usually code wrapped around a fixed model (this is what we also refer to as the "frozen" model) consisting of the system prompt, per-question formatting hints, the decoding budget, and the post-processing that normalizes raw generation into an answer. From now on, we call this code the *inference program*. In practice, it is often tuned by hand, one model and one benchmark at a time, and the results rarely transfer. We mention here that prompt optimizers such as OPRO (Yang et al., 2024), APE (Zhou et al., 2023), and EvoPrompt (Guo et al., 2024) automate part of the work, but they search only over the prompt string. As a consequence, they do not increase accuracy for perception as much, since they cannot shrink a potentially verbose answer, nor can they strip a trailing percent sign, and so accuracy is being compromised. Of course, one could in principle do retraining on chart-specialized models (Liu et al., 2023b; Masry et al., 2023), but this is quite expensive, and in fact, impossible when weights of the model are closed.

1.1 Our contribution

Our major finding is a proposed technique that completely avoids retraining, while significantly improving accuracy on all four benchmarks, thus answering the main question above affirmatively. The key technique is based on autonomous evolution, introduced originally for text-based Large Language Models (LLMs) by FunSearch (Romera-Paredes et al., 2024) and AlphaEvolve (Novikov et al., 2025).

Specifically, we can start with the initial inference program, which is a relatively small piece of Python code, and using autonomous evolution, we can improve it based on a cheap and reliable evaluator of the given responses. The edits are proposed by an LLM, which we call the *LLM-judge* (in the experiments we used Gemini-2.5-flash). We apply this idea to VLMs and report an increased accuracy on the 4 tested benchmarks, for 3 diversely-sized VLMs (up to 10B parameters).

Evolution of the inference program. Fig. 1 shows how we evolve the inference program. Starting from the model as is (this is the "frozen" model), each new generation points the LLM-judge to a few of the program's concrete failures and demands an improving edit in the inference code. The candidate programs live in a small pool ranked by training accuracy, and the next edit is made on the top-scorer. As is usual in such approaches, to limit overfitting, the final inference program that we output is the one that does best on a disjoint holdout window, that the search never optimized. To make this search practical, we use two safeguards: a static validator rejects malformed candidate programs before any GPU time is spent, and the model together with the VLM's decoding mode are fixed, so that every measured change in accuracy is attributable to the inference code (that we do the evolution on).

Below, we give more details on how we leverage the idea of autonomous evolution to improve (small) VLMs for the analysis of complex images like scientific charts.

- **Inference programs as the unit of optimization.** We formalize the executable inference program, instead of the prompt string or the model weights, as the object to be optimized for VLM chart question answering, and we build an evolutionary search framework over this space with an LLM as the mutation operator (Sec. 3).

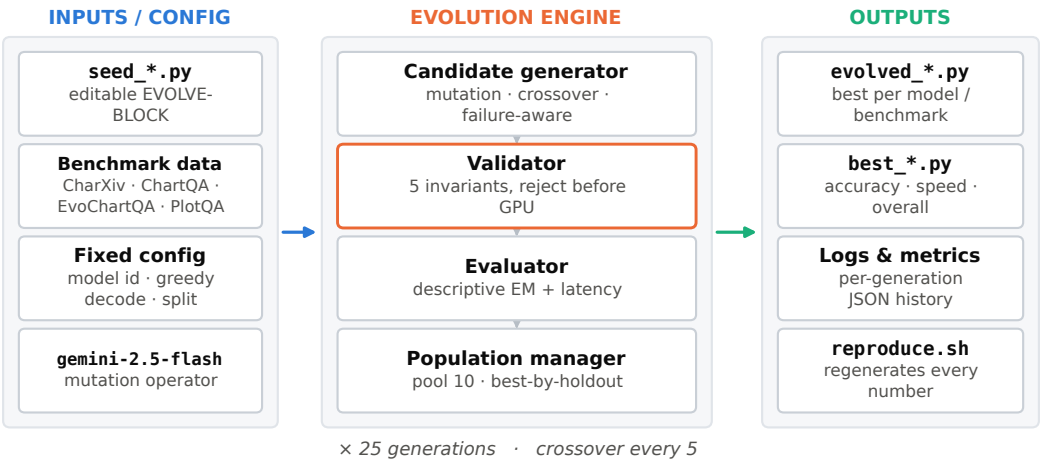

Figure 1: Our autonomous evolution framework for better VLMs. Fixed inputs (left) feed an evolution engine (middle) that outputs deployable inference programs (right). The model id, greedy decoding, and the train/holdout split never change during a run; only the code inside the `EVOLVE-BLOCK` of a candidate is being mutated. An LLM judge (in our case Gemini-2.5-flash) proposes edits, a validator rejects broken candidate programs before any GPU work, an evaluator scores survivor programs by exact-match accuracy, and a population manager keeps the best-by-holdout program. The same engine produces all twelve runs by swapping the seed program and the benchmark evaluator.

- **Safeguards that make the search trustworthy.** We introduce a validator that enforces five static invariants, and monitors GPU evaluation so that compute is spent only on valid candidates. This is then decoupled from a selection rule whose goal is to reduce overfitting.
- **Evaluation over 4 benchmarks and 3 models.** We finally evaluate on three differently sized open VLMs (Qwen3-VL-2B, InternVL3, SmolVLM2) and four chart benchmarks (CharXiv, ChartQA, EvoChartQA, PlotQA). We characterize when the method is helpful, when it is not, and what it costs (Sec. 4).

Benefits of our proposed method. In comparison to simple prompt optimization, our search space contains the token budget of each response and the post-processing rule of the candidate inference program. By using the LLM judge, and without the expensive retraining, we can perform a single edit to the inference program that later changes several of these together. In contrast to prompt optimization, our evolutionary search can fix errors that no re-wording could. Compared with retraining, the proposed method does not effectively change any weights, so it can be applied to any checkpoint that fits a "one-function" interface, including closed models. Comparing our method with hand-tuning, the search is automatic and reproducible: on Qwen3-VL-2B/ChartQA the evolved program matches a hand-tuned program in accuracy, and beats it on a combined accuracy-throughput score.

Experimental results. Figure 2 and Table 1 show a summary of the key results achieved by our proposed method. We report that across all twelve model-benchmark pairs, the evolved program improves descriptive exact-match accuracy over the native "frozen" baseline model. *The average accuracy over the 12 combinations rises from 16.6% to 41.3%, and the relatively largest gain is 55.2 percentage points (Qwen3-VL-2B on ChartQA, from 0.5% to 55.7%).* There are several instances where more robust models' errors are perceptual (SmolVLM2 on CharXiv, $+2.3$; InternVL3 on PlotQA, $+2.6$). Upon analyzing the results of Qwen3-VL-2B/ChartQA, the fastest evolved variant is about 20 points more accurate than the unevolved model and 3.8 times faster, because the edits performed by our method produce short and concise answers, thus needing less compute.

2 Related Work

Prior work in the area of VLMs has addressed the performance gap between models and humans on chart question answering through different approaches, such as modifying the model weights, prompts, or inference programs. While these methods offer specialized optimizations, they either

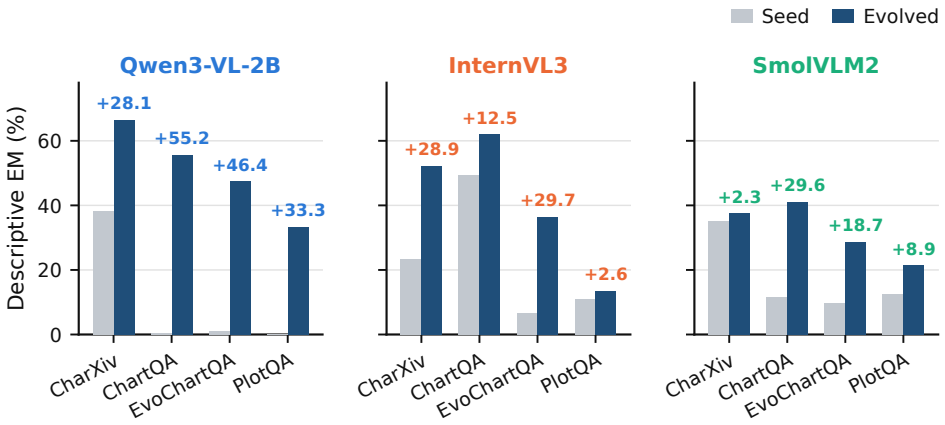

Figure 2: Seed vs. our evolved descriptive exact-match accuracy on all 12 model–benchmark pairs. Annotations give the absolute gain in percentage points by our method. As observed, every pair improves with the largest improvement being $+55.2$ percentage points. The largest jumps are on Qwen3-VL-2B, whose naive seed produced verbose, wrongly formatted answers; the smallest is SmolVLM2/CharXiv, whose seed was already clean (EM stands for Exact-Match).

	CharXiv		ChartQA		EvoChartQA		PlotQA	
Model	Seed	Evolved	Seed	Evolved	Seed	Evolved	Seed	Evolved
Qwen3-VL-2B	38.3	**66.4**	0.5	**55.7**	1.0	**47.4**	0.0	**33.3**
InternVL3	23.4	**52.3**	49.5	**62.0**	6.8	**36.5**	10.9	**13.5**
SmolVLM2	35.2	**37.5**	11.5	**41.1**	9.9	**28.6**	12.5	**21.4**

Table 1: Descriptive exact-match accuracy (%) of the seed and the evolved program for all 12 runs, computed on the full descriptive portion of each benchmark. Mean over runs: seed 16.6%, evolved 41.3%.

constrain the search space to text strings, operate within single-trajectory feedback loops or require modifying frozen model weights. In this section, we discuss these key research directions and how our work relates to them.

Chart understanding. Several benchmarks have been introduced for chart question answering, each focusing on different aspects of chart understanding. Chart question answering was established as the benchmark task in ChartQA (Masry et al., 2022), while PlotQA (Methani et al., 2020) introduced a large set of algorithmically generated questions, and EvoChartQA (Huang et al., 2025) expanded the space of question types in real-world charts. CharXiv (Wang et al., 2024) further focused on targeted scientific charts from arXiv papers and separated *descriptive* questions (evaluated via exact match) from open-ended *reasoning* questions. The exact match protocol makes the formatting as important as semantic accuracy: correct values are marked incorrect if they are wrapped in sentences or extra units. Existing work improves chart understanding by changing the model, through chart-specific pretraining MatCha (Liu et al., 2023b) and UniChart (Masry et al., 2023), or through intermediate representations DePlot (Liu et al., 2023a), which converts a chart to a table before reasoning. These approaches require modifying the underlying model through retraining or fine-tuning. In contrast, our approach leaves the model weights untouched and modifies only the code around inference, including the prompt, decoding budget, and output parser. This makes the approach applicable to frozen models, including closed-weight models that cannot be modified through methods such as DePlot or MatCha.

Inference-time and prompt optimization. Prompt optimization methods such as OPRO (Yang et al., 2024), APE (Zhou et al., 2023), and EvoPrompt (Guo et al., 2024) use an LLM to iteratively search over prompt strings, improving performance without model retraining. Since their search space is confined to prompt text, they cannot adjust decoding parameters, allocate dynamic token budgets, or add post-processing to normalize outputs into gradable answers. Other frameworks like DSPy (Khattab et al., 2024) and TextGrad (Yuksekgonul et al., 2025) optimize declarative modules

and text-valued parameters, and systems such as SGLang (Zheng et al., 2024) and Ayo (Tan et al., 2025) optimize application execution around fixed weights. Building on this idea that the application layer is an ideal optimization target, we go one step further by searching directly over the executable program itself.

Evolutionary and self-improving code search. The idea of evolving executable code dates back to genetic programming (Koza, 1992), but it became more practical with the rise of LLMs. FunSearch (Romera-Paredes et al., 2024) showed that an LLM can serve as a mutation operator, discovering new constructions in combinatorics, and AlphaEvolve (Novikov et al., 2025) extended this idea to larger algorithmic and systems problems while maintaining population diversity. Our work builds on this key insight that operating over code, rather than prompt strings or model weights, allows the search to make structural changes that prompt-only optimization cannot express. In parallel, Self-Refine (Madaan et al., 2023) and Reflexion (Shinn et al., 2023) showed that feedback can be used to improve outputs within a single trajectory, but they do not maintain a population or persist reusable code artifacts. Inspired by their use of concrete errors as feedback, we use a failure-aware mutation operator within a population-based framework. To our knowledge, this is the first work to apply LLM-driven code evolution to the inference pipelines of vision language models.

LLM serving systems. Existing LLM serving systems, such as Orca (Yu et al., 2022), vLLM (Kwon et al., 2023), and dLoRA (Wu et al., 2024), boost throughput by low-level optimizations like request batching, memory management, and adapter-aware scheduling. Our work takes a complementary approach: instead of modifying the underlying serving infrastructure, we optimize what the model is asked to do, how much it may generate, and how its output is parsed. Since our optimizations focus on program structure rather than serving infrastructure, our evolved programs can be deployed seamlessly on top of any of these existing systems.

3 Autonomous Evolution of VLMs

We start by giving a mathematical description for a fixed Vision Language Model and some notation used for the chart understanding task, the input question and the correct answer (we refer to it as the "gold" answer).

3.1 Problem setting and key idea

Let f be a fixed VLM and let a benchmark be a set of triples (x, q, a) of (chart image, question, gold answer). Each question carries an identifier for the type of question that it asks, labeled as qid (for example, "highest y-tick" or "number of series" etc.). An *inference program* p is a function $p(x, q, qid)$ mapping a specific triplet to a string. Function $p(\cdot)$ calls $f(\cdot)$ zero or more times, and returns a single string to be graded. Its descriptive accuracy in a set S is measured by the exact match $\mathrm{EM}_S(p) = \frac{1}{|S|} \sum_{(x,q,a)\in S} \mathbb{P}[\, p(x, q, qid) = a \,]$ after normalization of the benchmark. We are given a training window T and a disjoint holdout window H, and we seek a program that maximizes EM on questions it has never seen.

The key idea is that $p(\cdot)$ is a short Python module, and the undesired behavior that we want to fix is located within the program $p(\cdot)$ rather than in $f(\cdot)$. For example, a verbose answer that fails exact match is corrected not by re-wording the question, but by budgeting fewer tokens and by normalizing the raw string. Therefore, we make $p(\cdot)$ the object of our evolutionary search. Following techniques from FunSearch (Romera-Paredes et al., 2024) and AlphaEvolve (Novikov et al., 2025), the search is an evolutionary loop in which an LLM is the so-called "mutation" operator, EM_T is the "fitness," and a population of programs is maintained across generations.

3.2 Evolution search over candidate programs

In the context of our evolution search, a candidate is a single Python module, where only the code between the markers `# EVOLVE-BLOCK-START` and `# EVOLVE-BLOCK-END` is being edited. Everything outside these markers is fixed, and this allows for a fast search. The changes are performed in one function, `vlm_inference(image_path, question, qid)` $\rightarrow$ `str`, which

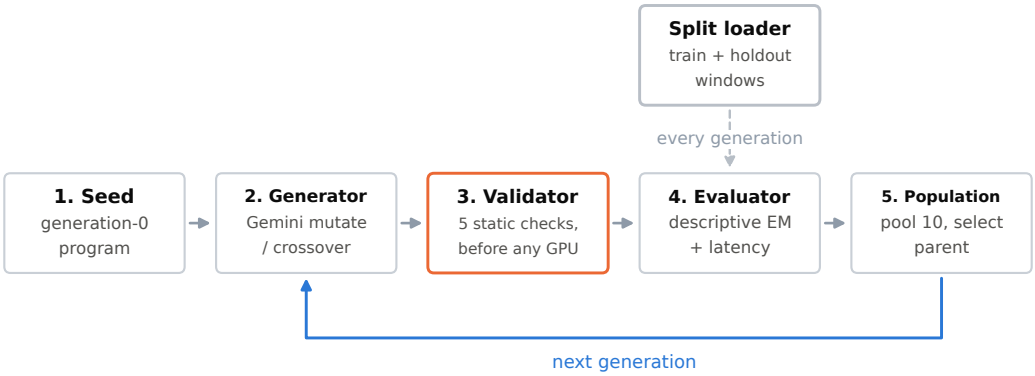

Figure 3: The evolution loop. In every generation, the split loader feeds the train and holdout windows to the evaluator. The LLM-judge proposes edits, the validator checks them before any GPU work is performed, and the population manager selects the next parent and writes the best-by-holdout candidate to disk. Every five generations, a crossover step combines two strong parents into one child.

is responsible for mapping a plot and its question to an answer. Inside the editable code, our optimizer may change the system prompt, per-*qid* formatting hints, per-*qid* token budgets, and the post-processing that normalizes the raw output. Throughout our evolution search, both the model name and greedy decoding (`do_sample=False`) are fixed by the validator and never change. We emphasize that the fixed decoding is what allows the generations to be reproducible, so that any accuracy difference between two generations is due to the code edit and not sampling noise (of the VLM).

3.3 The evolution loop

In this section, we give descriptions of our evolution loop. Each generation performs five steps as shown in Fig. 3. The final outcome of this process is the best candidate inference program found during our search.

1. Seed. Generation 0 is a deliberately naive baseline program: a plain prompt, a generous token budget, and no post-processing. Its accuracy is low by design, so that the search starts from the same place a practitioner would.

2. Candidate generator (mutation operator). The mutator is an LLM, Gemini 2.5 Flash (Comanici et al., 2025), in all experiments. As typical in evolution approaches, *mutation* takes one strong parent and makes a single targeted edit, such as tightening the prompt, adjusting a per-*qid* hint, changing a token budget, or rewriting post-processing. *Crossover*, performed every five generations, takes two strong parents and synthesizes a child that keeps the better rule for each question type. A consequence of our proposed method is that the operator is *failure-aware*: six concrete failures of the parent are sampled, and each triple (question, model answer, gold answer) is inserted into the prompt, so the LLM-judge edits toward what is actually breaking rather than rewriting at random. The six are drawn from *different* question types rather than being the six worst cases (we observed that concentrating on just one failing type would bias the mutator towards a narrow fix that only helps that question type at the expense of others).

3. Validation before compute. LLM-generated Python modules frequently break at run time through invalid chat formatting, incorrect imports that may fail, or false function signatures (Tambon et al., 2025; Liu et al., 2023c). Because loading the VLM and running an evaluation is an expensive step, the validator's role is to reject bad candidate programs by performing cheap static checks before any GPU is used. Here, we enforce five invariants to check validity of Python modules (Table 2).

4. Evaluator. The evaluator runs a top-scoring candidate program over the training window T and the holdout window H and reports four quantities. *Descriptive exact match* is the primary signal

#	Invariant	Check
1	Markers	`EVOLVE-BLOCK` present
2	Deterministic decode	no `do_sample=True`
3	Model unchanged	backbone id preserved
4	Signature	`vlm_inference(...)` intact
5	Syntax	`ast.parse()` succeeds

Table 2: The five validator invariants. Checks 1–4 are cheap string tests run before the costlier syntax and import stage; the whole gate runs before any GPU work.

and the only quantity that drives selection. *Reasoning exact match* is computed as a proxy and is never used as fitness, because the official reasoning metric is open-ended and exact match is the wrong metric for it. In our experiments, we keep track of the *latency* (hence throughput) and *errors* (exceptions or empty generations), but these do not drive the search.

5. Population manager and selection. The population is a pool of at most ten programs sorted by their training accuracy. As usual in evolution search, the weakest of them are pruned in each generation, while parent selection is done via a tournament of three (80% of the time) and a draw from the weaker half of the pool (20% of the time), thus creating a balance between exploitation and exploration. Crucially, while parents are chosen based on *training* accuracy, the candidate written to memory in each generation is the one with the best *holdout* accuracy. Decoupling the two roles prevents overfitting. In our experiments, a run lasts 25 generations and produces the best-by-holdout program together with a JSON history of every event (see Appendix B).

4 EXPERIMENTS

In this section, we report the experimental setup (the code and files for which can be found in the anonymous link: `https://anonymous.4open.science/r/vlm-evolution-supp-6631/README.md`), observations and the experimental results obtained by the proposed technique to improve the VLMs. Our main result is that we significantly increased the VLM accuracy for each model and benchmark pair, while being faster than previous approaches (see Fig. 4).

Models. We evolve inference programs for three popular and compact open-source VLMs: Qwen3-VL-2B-Instruct (Bai et al., 2025), InternVL3 (Zhu et al., 2025), and SmolVLM2 (Marafioti et al., 2025). Qwen3-VL-2B is the strongest of the three on chart tasks; InternVL3 is included for its strong document and chart performance; SmolVLM2 represents a smaller, efficiency-oriented model. All three are run in float16 through HuggingFace Transformers (Wolf et al., 2020) with greedy decoding.

Datasets. We use four extensive chart question-answering benchmarks: CharXiv (Wang et al., 2024) (scientific charts from arXiv papers), ChartQA (Masry et al., 2022) (real-world charts with human-written and machine-generated questions), EvoChartQA (Huang et al., 2025) (a wider range of question types), and PlotQA (Methani et al., 2020) (large-scale algorithmically generated questions). Each benchmark's descriptive questions are split into a training window, used to compute fitness, and a disjoint holdout window, used only to choose the saved program. Reasoning questions are evaluated but never used for selection. For Qwen3-VL-2B/ChartQA the windows are 102 descriptive plus 51 reasoning training samples and 26 descriptive plus 13 reasoning holdout samples. Together this gives $3 \times 4 = 12$ evolution runs.

Other methods. Our primary comparison is against the *seed* program: baseline inference with no hints and no post-processing, evaluated end-to-end through the same pipeline as the evolved programs. For the Qwen3-VL-2B/ChartQA analysis, we add a hand-tuned program as a stronger human-designed baseline, and we report the three deployment variants that our pipeline extracts from the final pool (`best_accuracy`, `best_speed`, `best_overall`). We do not run prompt-only optimizers such as OPRO or EvoPrompt as separate baselines; their search space is a strict

Question	Gold	Seed output ($\times$)	Evolved
Highest labeled y-tick?	**60**	"The y-axis label is … the highest tick value is 58 …"	**60**
Year of largest blue–green gap?	**2018**	"… we calculate: 2005: $\lvert 43-46\rvert=3$, 2016 …"	**2018**
% supporting TikTok ban, Sept.	**38**	"… the value is 28. Therefore, 28%"	**38**
% with prior military service, 2023	**17**	"16%"	**17**

Table 3: Examples of incorrect instances (abbreviated). The seed baseline perceives incorrectly and often returns a long, verbose response; the evolved program returns the exact answer. Wrong values in the seed output are underlined. Examples are drawn from Qwen3-VL-2B, InternVL3 and SmolVLM2 runs.

subset of ours, and the qualitative analysis in Table 3 shows that the gains come predominantly from edits outside the prompt.

Experimental setup. The primary metric is descriptive exact-match accuracy after each benchmark's normalization. Speed is recorded in queries per second for every run but is not a selection signal. Each run performs 25 generations with a pool of at most ten programs, crossover every five generations, and up to six failure examples per mutation prompt; Gemini 2.5 Flash is the mutator throughout. Runs were performed on Google Colab A100 GPUs; one run takes on the order of a few hours on a single GPU. Final accuracies in Table 1 are computed over the full descriptive portion of each benchmark, not only the holdout window.

4.1 Results

Our evolution search improves every configuration. Table 1 and Fig. 2 report baseline and evolved accuracy for all twelve runs. Mean accuracy rises from 16.6% to 41.3%, a mean gain of 24.7 points, and every run improves. The size of the gain depends strongly on the type of error the seed makes. Where the seed identifies the right answer but returns it in an ungradable form, the gain is large: Qwen3-VL-2B goes from 0.5% to 55.7% on ChartQA ($+55.2$) and from 1.0% to 47.4% on EvoChartQA ($+46.4$). Where the seed's outputs are already cleanly formatted, there is little room for the search: SmolVLM2 on CharXiv gains 2.3 points and InternVL3 on PlotQA gains 2.6. Ranking the twelve gains (Appendix C) shows this split clearly.

Errors of the baseline models. Table 3 shows a few examples of incorrect responses of the baseline VLMs. In each, the seed's answer contains the incorrect value and is not gradable: it has a long explanation. In contrast, the evolved program returns the standardized string directly. These examples support the claim that the gain does not come only from better visual understanding; the search is improving the *structure* of the inference program, and in particular how the answers are formatted and extracted.

Our evolution search is fast and accurate. Because the pipeline extracts variants tuned for accuracy, speed, and a balance of the two, we examine Qwen3-VL-2B on ChartQA in detail. Table 4 and Fig. 4 report all six programs on a window of 128 descriptive and 64 reasoning samples, together with an aggregate score, the harmonic mean of descriptive accuracy and throughput capped at 10 queries per second, floored to zero below 30% accuracy (Appendix A). This score is used only for ranking deployment variants; the search itself optimizes descriptive exact match alone. On this window, the seed reaches 38.3% at 0.33 queries per second. In contrast, our `best_speed` variant reaches a better accuracy of 58.6%, while operating at a higher throughput of 1.24 queries per second. This is about 20 points more accurate and 3.8$\times$ faster, thus making for an improved VLM overall.

Moreover, even though one might expect exact-match gains to come at a cost for latency, through longer prompts or more deliberate reasoning, we observe the opposite behavior. This is because the edits performed by our evolution approach create brief answers that are more easily gradable, and also remove unnecessary generation. Thus, they are overall much faster to produce.

Program	Desc. EM	Rsn. EM	QPS	Overall
seed (starting)	38.3%	4.7%	0.33	0.061
hand-tuned (human)	56.3%	17.2%	1.20	0.198
evolved	58.6%	9.4%	0.78	0.138
best_accuracy	58.6%	10.9%	0.85	0.148
best_speed	**58.6%**	14.1%	**1.24**	**0.205**
best_overall	57.0%	9.4%	1.22	0.201

Table 4: Qwen3-VL-2B / ChartQA, all six programs (128 descriptive and 64 reasoning samples). `best_speed` is both the fastest and the highest on the aggregate score, beating the hand-tuned baseline.

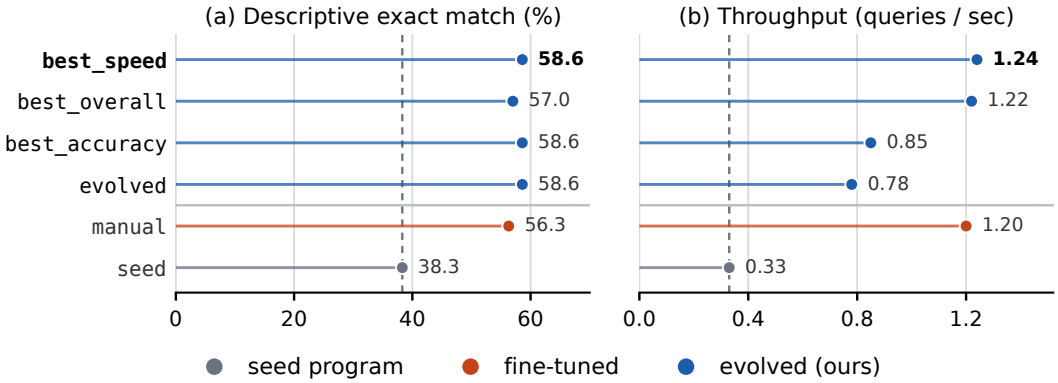

Figure 4: Accuracy vs. throughput, Qwen3-VL-2B/ChartQA. Evolution moves the program up and to the right of the seed; `best_speed` dominates on both axes.

5 Conclusion

In this study, we demonstrated that VLM inference for scientific chart question answering can be substantially improved by proposing an evolution-based method that edits the inference program rather than the prompt alone. For three different VLMs, we found improvements across four benchmarks without retraining the models. For all 12 model-benchmark pairs, we improved the mean descriptive exact-match accuracy from 16.6% to 41.3%, with the largest gain of 55.2 points. The method combines an LLM-based mutation operator, GPU evaluation gated by a static validator, and a best-by-holdout selection rule. The fastest evolved program on Qwen3-VL-2B/ChartQA was 3.8 times faster than the seed baseline program and simultaneously more accurate than a hand-tuned baseline. We observe that our method performs most effectively when the correct answer is wrapped in verbose reasoning in the model output. We observed that it was relatively less effective with perceptual failures and in cases with no model response.

AI Use Statement

In this work, we used generative AI tools to implement methods, in the specific sense that Gemini 2.5 Flash is the mutation operator of our method and writes the candidate inference programs that we evaluate (Sec. 3); this use is the subject of the paper, every program the model produced was validated automatically and evaluated empirically, and the evolved programs are research artifacts of this work. We have not used generative AI tools to develop theoretical models or conceptual frameworks, to propose or refine hypotheses, to design or provide feedback on research methodology or experiments, to interpret results, or to support qualitative or thematic data analysis. Generating synthetic datasets, formulating or proving mathematical claims, translation, and cleaning or reformatting datasets are not applicable to this work. Additionally, we used generative AI tools to edit the paper to improve readability, to suggest a structure for the paper, to format references, and to check compliance with the formatting instructions. We have reviewed all AI-assisted work. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

Reproducibility Statement

All runs use greedy decoding with the model id fixed by the validator, so re-running a saved program reproduces its outputs exactly. Sec. 3 specifies the candidate contract, the validator invariants (Table 2), the mutation and crossover operators, and the selection rule; Sec. 4 specifies the models, benchmarks, split sizes, generation count, pool size, crossover interval, and hardware. Appendix B describes the code layout and the per-generation JSON history, and Appendix A defines the aggregate score used in the deep dive. The anonymous source code, seed programs, evolved programs, and evolution logs for all twelve runs can be found at `https://anonymous.4open.science/r/vlm-evolution-supp-6631/README.md`.

References

Shuai Bai, Yuxuan Cai, Ruizhe Chen, Keqin Chen, Xionghui Chen, Zesen Cheng, Lianghao Deng, Wei Ding, Chang Gao, Chunjiang Ge, et al. Qwen3-VL technical report. *arXiv preprint arXiv:2511.21631*, 2025.

Gheorghe Comanici, Eric Bieber, Mike Schaekermann, Ice Pasupat, Noveen Sachdeva, Inderjit Dhillon, Marcel Blistein, Ori Ram, Dan Zhang, Evan Rosen, et al. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *arXiv preprint arXiv:2507.06261*, 2025.

Qingyan Guo, Rui Wang, Junliang Guo, Bei Li, Kaitao Song, Xu Tan, Guoqing Liu, Jiang Bian, and Yujiu Yang. Connecting large language models with evolutionary algorithms yields powerful prompt optimizers. In *International Conference on Learning Representations (ICLR)*, 2024.

Muye Huang, Han Lai, Xinyu Zhang, Wenjun Wu, Jie Ma, Lingling Zhang, and Jun Liu. EvoChart: A benchmark and a self-training approach towards real-world chart understanding. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pp. 3680–3688, 2025. doi: 10.1609/aaai.v39i4.32383.

Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. DSPy: Compiling declarative language model calls into self-improving pipelines. In *International Conference on Learning Representations (ICLR)*, 2024.

John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with PagedAttention. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2023.

Fangyu Liu, Julian Martin Eisenschlos, Francesco Piccinno, Syrine Krichene, Chenxi Pang, Kenton Lee, Mandar Joshi, Wenhu Chen, Nigel Collier, and Yasemin Altun. DePlot: One-shot visual language reasoning by plot-to-table translation. In *Findings of the Association for Computational Linguistics (ACL)*, 2023a.

Fangyu Liu, Francesco Piccinno, Syrine Krichene, Chenxi Pang, Kenton Lee, Mandar Joshi, Yasemin Altun, Nigel Collier, and Julian Martin Eisenschlos. MatCha: Enhancing visual language pretraining with math reasoning and chart derendering. In *Proceedings of the Association for Computational Linguistics (ACL)*, 2023b.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by ChatGPT really correct? rigorous evaluation of large language models for code generation. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023c.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. Self-refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

Andres Marafioti, Orr Zohar, Miquel Farré, Merve Noyan, Elie Bakouch, Pedro Cuenca, Cyril Zakka, Loubna Ben Allal, Anton Lozhkov, Nouamane Tazi, Vaibhav Srivastav, et al. SmolVLM: Redefining small and efficient multimodal models. *arXiv preprint arXiv:2504.05299*, 2025.

Ahmed Masry, Do Xuan Long, Jia Qing Tan, Shafiq Joty, and Enamul Hoque. ChartQA: A benchmark for question answering about charts with visual and logical reasoning. In *Findings of the Association for Computational Linguistics (ACL)*, 2022.

Ahmed Masry, Parsa Kavehzadeh, Xuan Long Do, Enamul Hoque, and Shafiq Joty. UniChart: A universal vision-language pretrained model for chart comprehension and reasoning. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2023.

Nitesh Methani, Pritha Ganguly, Mitesh M. Khapra, and Pratyush Kumar. PlotQA: Reasoning over scientific plots. In *IEEE Winter Conference on Applications of Computer Vision (WACV)*, 2020.

Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, and Swarat Chaudhuri. AlphaEvolve: A coding agent for scientific and algorithmic discovery. Technical report, Google DeepMind, 2025.

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M. Pawan Kumar, Emilien Dupont, Francisco J. R. Ruiz, Jordan S. Ellenberg, Pengming Wang, Omar Fawzi, Pushmeet Kohli, and Alhussein Fawzi. Mathematical discoveries from program search with large language models. *Nature*, 625:468–475, 2024.

Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

Florian Tambon, Arghavan Moradi Dakhel, Amin Nikanjam, Foutse Khomh, Michel C. Desmarais, and Giuliano Antoniol. Bugs in large language models generated code: An empirical study. *Empirical Software Engineering*, 30(3):65, 2025. doi: 10.1007/s10664-025-10614-4.

Xin Tan, Yimin Jiang, Yitao Yang, and Hong Xu. Towards end-to-end optimization of LLM-based applications with Ayo. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 1302–1316, 2025. doi: 10.1145/3676641.3716278.

Zirui Wang, Mengzhou Xia, Luxi He, Howard Chen, Yitao Liu, Richard Zhu, Kaiqu Liang, Xindi Wu, Haotian Liu, Sadhika Malladi, Alexis Chevalier, Sanjeev Arora, and Danqi Chen. CharXiv: Charting gaps in realistic chart understanding in multimodal LLMs. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. Transformers: State-of-the-art natural language processing. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing: System Demonstrations (EMNLP)*, 2020.

Bingyang Wu, Ruidong Zhu, Zili Zhang, Peng Sun, Xuanzhe Liu, and Xin Jin. dLoRA: Dynamically orchestrating requests and adapters for LoRA LLM serving. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2024.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V. Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers. In *International Conference on Learning Representations (ICLR)*, 2024.

Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for transformer-based generative models. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2022.

Xiang Yue, Yuansheng Ni, Kai Zhang, Tianyu Zheng, Ruoqi Liu, Ge Zhang, Samuel Stevens, Dongfu Jiang, Weiming Ren, Yuxuan Sun, et al. MMMU: A massive multi-discipline multimodal understanding and reasoning benchmark for expert AGI. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 9556–9567, 2024.

Mert Yuksekgonul, Federico Bianchi, Joseph Boen, Sheng Liu, Zhi Huang, Carlos Guestrin, and James Zou. TextGrad: Automatic "differentiation" via text. *Nature*, 639:609–616, 2025.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. SGLang: Efficient execution of structured language model programs. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024.

Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. Large language models are human-level prompt engineers. In *International Conference on Learning Representations (ICLR)*, 2023.

Jinguo Zhu, Weiyun Wang, Zhe Chen, Zhaoyang Liu, Shenglong Ye, Lixin Gu, Hao Tian, Yuchen Duan, Weijie Su, Jie Shao, et al. InternVL3: Exploring advanced training and test-time recipes for open-source multimodal models. *arXiv preprint arXiv:2504.10479*, 2025.

A Aggregate score for the deep dive

For ranking the deployment variants in Table 4 we use

$$\text{OVERALL} = \text{harmonic_mean}(\text{desc_acc},\ \min(\text{QPS}/10,\ 1)),$$

set to zero when descriptive accuracy falls below 30%. The harmonic mean penalizes programs that are strong on only one dimension: a program should not rank highly if it is fast but mostly wrong, or accurate but too slow to be useful. The 30% floor reflects the same practical concern, since a chart reader with very low accuracy has little deployment value regardless of speed. The score is used only for reporting; the evolutionary search optimizes descriptive exact match alone.

B Code structure and logging

The implementation is roughly six Python files, organized so that the engine is reused unchanged across runs:

- `seed_*.py`: the generation-0 program, naive inference with no hints or post-processing.
- `evolve_<benchmark>.py`: the engine, comprising the population manager, parent selection, mutation and crossover, validation, and the generation loop.
- `evolve_common.py`: validation helpers, including reading the required model name from the seed and rejecting any model swap before GPU time.
- `evaluate.py` / `evaluate_<benchmark>.py`: load images and answers, run a candidate, compute accuracy and timing.
- `evolved_<benchmark>.py`: the winner after 25 generations, selected on questions it never saw during evolution.
- `best_accuracy.py`, `best_speed.py`, `best_overall.py`: three deployment variants drawn from the final pool.

Inference uses PyTorch with HuggingFace Transformers (Wolf et al., 2020) (`AutoModelForImageTextToText` / `AutoProcessor`) in float16 single-query mode. The mutator is called through the Google Generative AI SDK. Every generation prints the parent, the child's train and holdout accuracy, and the current best, and a JSON history records each event (seed, improvement, no_improvement, crossover, mutation_failed) with accuracies and timing. A representative excerpt:

```
[Gen 7/25] Parent: 56.9%/57.7% (gen 5)
  Child: train=58.6% holdout=61.5%
  Best by holdout: 61.5% -> saved
[Gen 10/25] [CROSSOVER] A:58.6/61.5
                    B:55.5/57.7
  Child: train=57.0% holdout=61.5%
  (best-by-train != best-by-holdout)
```

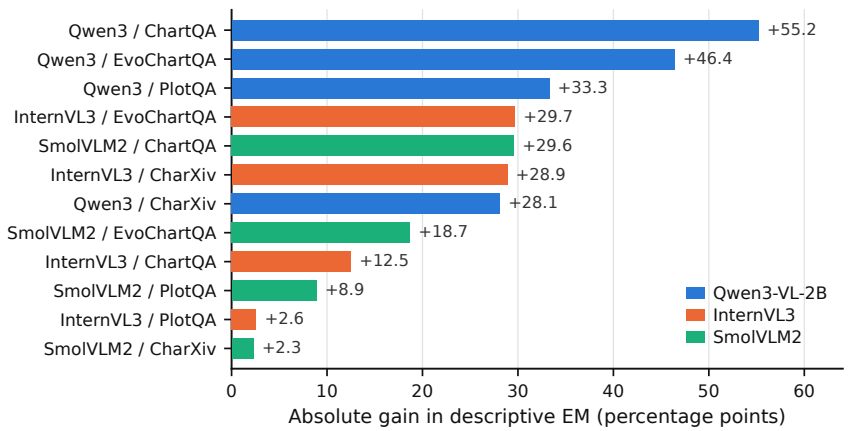

Figure 5: Absolute gain in descriptive exact match for each of the 12 runs, sorted. Color denotes model. Gains cluster by whether the seed's failure was one of *formatting* (large, recoverable) or *perception* (small headroom).

C PER-RUN GAINS, SORTED

Figure 5 ranks the twelve runs by their absolute gain in descriptive exact match.

D ENGINEERING FRICTIONS

Several candidate models were excluded for practical reasons. Some gated checkpoints, such as MiniCPM-V-2.6 and PaliGemma2, were delayed by access approvals. Others, including Molmo, Pixtral, and Phi-3.5-Vision, did not load correctly under our unified inference template. We also tested and discarded two efficiency strategies. Request batching for Qwen3-VL-2B ran into a right-padding issue in its `Qwen2VLProcessor` that capped accuracy at 29.7%. INT4 quantization ran at only 0.04 queries per second on a T4 GPU, roughly 75 times slower than float16, and was not pursued. These are not algorithmic failures, but they reflect the practical difficulty of building a uniform evaluation harness across heterogeneous VLMs.