What Makes Agentic Evolution Work?

Seongryong Jung
Chung-Ang University
jungsr1116@cau.ac.kr

Abstract

LLM-driven evolutionary code optimization improves programs through repeated mutation, evaluation, and selection, yet it remains unclear which properties of the surrounding optimization loop make this process effective. We introduce PRISMEVOLVE, a controlled framework for studying candidate selection, evaluation feedback, initialization, and reuse of prior search experience in VLM inference optimization. We also construct interpretable inference programs through systematic manual optimization. Our experiments show that cumulative improvement depends on more than inheritance alone. Promising descendants must be selectively propagated, and richer evaluation feedback provides its largest benefit when the search policy can repeatedly act on it. We further find that initial accuracy and evolvability can diverge substantially, as a structured manual start improves much more under continued evolution than a higher-accuracy previously evolved program. How prior search experience is carried forward also matters, with lineage-specific ancestry memory helping in our setting while randomly sampled archive inspiration does not consistently help. These results show that effective agentic evolution depends not only on generating useful mutations, but also on how improvements are identified, propagated, retained, and built upon.

1 Introduction

Large language models are increasingly used not only to generate programs once, but to improve them through repeated cycles of mutation, evaluation, and selection [4, 5, 7, 8]. In these systems, an LLM proposes modifications to an existing program, an executable evaluator measures their quality, and successful candidates can become the basis for subsequent changes [4, 7, 8]. This iterative process creates the possibility of accumulating improvements over multiple generations. However, it also introduces a broader design problem because performance depends not only on the mutation model, but also on how the surrounding optimization loop decides what to expand, what feedback to expose, what prior search experience to reuse, and where the search begins.

This paper studies these questions through visual-language-model (VLM) inference optimization. We treat the inference pipeline surrounding a fixed VLM as the program to be improved. Such a program includes choices about execution, prompting and response generation, visual preprocessing, and output handling, all of which can affect both task accuracy and execution cost [2, 3]. This setting provides a concrete but nontrivial testbed for studying iterative program improvement because candidate programs are directly executable, their quality can be measured automatically, and useful modifications may need to accumulate across multiple search steps.

To study the optimization loop in a controlled manner, we introduce PRISMEVOLVE, an LLM-driven evolutionary code optimization framework that separates candidate selection, evaluation feedback, and reuse of prior search experience. Given a starting program and an executable evaluator, PRISMEVOLVE repeatedly selects a parent, constructs a mutation context, generates and validates a code patch, evaluates the resulting child, and stores it in a program archive. By varying individual

Preprint.

components while holding the mutation model, evaluation budget, and task fixed, the framework allows us to study which mechanisms help useful changes accumulate during evolution.

Before running these experiments, we construct interpretable starting programs for Qwen3-VL-2B-Instruct and Qwen3-VL-2B-Thinking. We systematically optimize the execution backend and batching strategy, response structure and generation budget, and image resolution. The resulting programs provide competitive manually optimized baselines while also exposing explicit components that can later become targets for evolutionary modification.

Our experiments reveal several patterns about when iterative evolution is effective. First, what gets inherited matters more than inheritance itself. Allowing changes to propagate provides little benefit when the search policy does not consistently carry forward promising descendants. Second, richer diagnostic feedback consistently outperforms aggregate feedback, with substantially larger gains when the search policy repeatedly selects and refines promising descendants. Third, initial accuracy does not determine evolvability. When evolution is restarted from two strong starting points, a manually optimized program improves substantially more than a higher-accuracy program produced by a previous evolutionary run, eventually reaching a higher endpoint. We additionally find that, in our setting, lineage-specific search experience can be useful, whereas randomly sampled experience from elsewhere in the archive does not provide a consistent benefit. The main search-policy pattern also transfers from Qwen3-VL-2B-Instruct to Qwen3-VL-2B-Thinking, and major evolutionary gains persist on a disjoint validation set containing different charts.

The principal contributions of this paper are as follows:

- We introduce PRISMEVOLVE, a controlled framework for studying how candidate selection, evaluation feedback, and reuse of search experience shape LLM-driven program evolution.
- We construct interpretable VLM inference programs through systematic manual optimization of execution, response generation, and visual preprocessing, providing challenging starting points with explicit components for subsequent evolution.
- Through controlled experiments, we identify how selection, feedback, initialization, and search-experience reuse shape sustained evolutionary improvement, and validate the resulting search procedure across a second target VLM and a disjoint validation split.

2 Related Work

Evolutionary Code Optimization with LLMs Evolutionary code optimization combines language-model generation with iterative evaluation and selection. An LLM generates candidate programs by modifying one or more existing solutions, while an executable evaluator measures their quality under task-specific objectives. Selected candidates inform subsequent mutations, forming a repeated cycle of variation, evaluation, and selection [4–9]. Unlike one-shot code generation, this process can use evidence from previously evaluated programs to guide later proposals. Its performance therefore depends on both the mutation model and the design of the surrounding optimization loop. Prior work primarily demonstrates that LLM-guided evolutionary search can produce strong programs, whereas our focus is on isolating which components of the surrounding optimization loop make iterative improvement accumulate.

Optimization of VLM Inference Pipelines VLM performance depends not only on model weights but also on the surrounding inference program [2, 3]. Input preprocessing, prompt construction, generation settings, and output processing can affect both task accuracy and execution cost. These components therefore form a practical optimization space around a fixed VLM. In this work, we treat the inference pipeline itself as the program to be improved, either manually or through evolutionary search. Prior work studies how inference-pipeline choices affect VLM accuracy and efficiency, whereas we use this optimization space to study how such program changes can be accumulated through iterative evolution.

3 PRISMEVOLVE

PRISMEVOLVE is an LLM-driven evolutionary code optimization system for studying how design choices affect iterative program improvement. Given an initial program, a task specification, and an

executable evaluator, the system repeatedly generates, evaluates, and selects program modifications while maintaining a program archive. Figure 1 illustrates this optimization loop.

At each iteration, the system selects a parent from the archive and constructs a mutation context from the parent program, evaluation feedback, and optional information from previous search. The mutation LLM generates a code patch, which is applied to the parent and validated against the required interface. The resulting child is evaluated and added to the archive, and the process continues until the evaluation budget is exhausted.

The arrows in Figure 1 organize our analysis. *Programs to improve* concerns parent selection and population management (Section 3.1). *Quality scores and other feedback* concerns the granularity of evaluator information returned to the mutation model (Section 3.2). *Past trials and ideas* concerns whether the prompt reuses information accumulated during search (Section 3.3). Through controlled comparisons, we examine how these choices affect search trajectories and the quality and efficiency of the programs discovered.

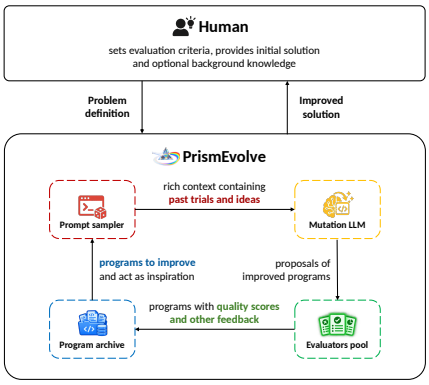


Figure 1: PRISMEVOLVE's evolutionary loop.

3.1 Candidate Selection and Population Management

PRISMEVOLVE stores evaluated programs in an archive and maintains an active population of candidates available for further modification. A search policy determines which parent is expanded and how the active population is updated. Separately, an evaluator-defined selection objective ranks candidates, allowing the same search policy to prioritize different properties of a program. We consider four search policies. *Fixed* repeatedly expands the initial program, providing a non-hereditary best-of-N baseline. *Greedy* repeatedly expands the highest-ranked candidate found so far. *Beam* maintains a bounded set of highly ranked candidates and expands them in successive rounds. *FIFO* expands eligible candidates in insertion order and appends their children to the end of the queue. These policies differ in selection pressure, the breadth of active exploration, and whether improvements can accumulate through repeated inheritance. The selection objectives and population settings used in our experiments are specified in Section 4.

3.2 Evaluation Feedback

Executable evaluation serves two distinct roles in PRISMEVOLVE. Evaluator measurements are used to rank candidates under the selection policy, while selected evaluation information may also be included in the mutation context to guide subsequent proposals. *Aggregate feedback* exposes program-level measurements and execution status. *Diagnostic feedback* additionally exposes instance-level outcomes that indicate where the current program succeeded or failed. Both modes are derived from the same evaluation records, isolating the effect of feedback granularity on mutation generation. The concrete feedback fields used in our experiments are specified in Section 4.

3.3 Reuse of Search Experience

PRISMEVOLVE reuses prior search experience through program inheritance and additional mutation context. Each child is produced by applying a generated patch to its selected parent. When descendants are subsequently selected as parents, modifications can accumulate across generations. *Ancestry memory* augments the mutation context with an ordered record of the selected parent's direct lineage. This record describes earlier modifications and their observed performance changes, providing information about how the current program was reached without summarizing sibling branches. *Archive inspiration* retrieves an eligible evaluated program other than the current parent as an additional reference. The reference program and its available evaluation information are included in the mutation context, but the generated patch is applied only to the selected parent.

4 Experimental Design

4.1 Task and Evaluation

We study evolutionary code optimization through chart question answering on CharXiv [10], a benchmark constructed from figures in scientific papers. The benchmark includes descriptive questions that require retrieving visually presented information and reasoning questions that require interpreting or combining multiple elements of a chart. We optimize the vlm inference function for Qwen3-VL-2B-Instruct and Qwen3-VL-2B-Thinking [1].

We evaluate each program using answer accuracy and end-to-end runtime. Accuracy measures task performance, whereas runtime includes the complete execution of the evaluated inference pipeline. Evolutionary search uses a fixed optimization set of 128 examples. We additionally evaluate representative frozen programs on a disjoint 128-example validation set containing different charts; this split is not used to generate mutation feedback and is used only for final program comparison and submission selection.

4.2 Experimental Factors

We organize the experimental conditions along four dimensions: candidate selection and population management, evaluation feedback, initialization, and reuse of search experience.

Candidate selection and population management. We configure candidate selection through a search policy and a selection objective. We compare Fixed, Greedy, Beam, and FIFO as the search policies described in Section 3.1. The selection objective is either accuracy-first or speed-first. Accuracy-first prioritizes higher accuracy and uses lower runtime to break ties, whereas speed-first prioritizes lower runtime and uses higher accuracy to break ties. Both objectives use the same mutation prompt, which asks the model to improve accuracy and runtime. They differ only in which evaluated programs are preferred as the starting points for subsequent mutations.

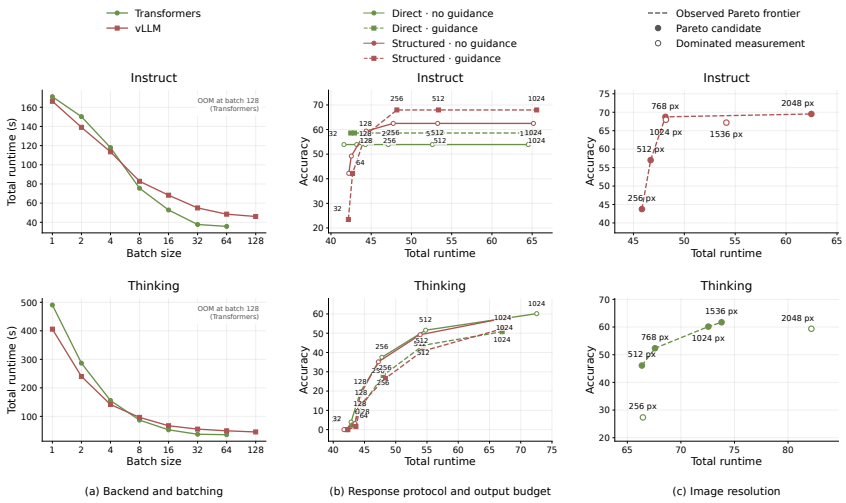
Evaluation feedback. We compare two levels of evaluation information exposed to the mutation model. Aggregate feedback includes the candidate's accuracy, runtime, and execution status. Diagnostic feedback additionally includes, for each evaluation example, the model prediction, reference answer, and correctness. Both conditions use the same evaluator and candidate-ranking procedure, differing only in the information included in the mutation prompt.

Initialization. We vary the starting program among a simple reference implementation, a manually optimized implementation, and a previously evolved program. This comparison examines whether starting quality alone determines evolvability, which we define as how readily a program yields further improvements under a fixed search configuration and mutation budget.

Reuse of search experience. We vary two forms of additional mutation context. Ancestry memory includes the mutation intent, code change, and performance change for each ancestor along the selected parent's lineage. Archive inspiration includes the code and evaluation result of an eligible archive program outside that lineage, selected at random.

4.3 Common Experimental Setup

Unless otherwise specified, our evolution experiments use Qwen3-VL-2B-Instruct as the target VLM and Gemini 3.6 Flash with medium reasoning as the mutation model. Each run permits up to 36 mutation attempts, organized into 12 parent-expansion steps with three children generated per selected parent. Target-model inference uses greedy decoding throughout all experiments. To isolate the effect of each experimental variable, we keep the mutation model, optimization set, mutation budget, and evaluation procedure fixed across the corresponding comparisons. Full prompts, generation settings, validation procedures, and policy-specific population parameters are provided in Appendix A.



(a) Backend and batching (b) Response protocol and output budget (c) Image resolution

Figure 2: Manual optimization of Qwen3-VL-2B-Instruct and Thinking. (a) Backend and batch-size comparison. (b) Accuracy–runtime trade-offs across response protocols and maximum output lengths. (c) Image-resolution comparison under selected response settings. Runtime includes initialization and evaluation over 128 examples; accuracy is reported as a percentage.

5 Constructing a Strong Initialization

Before studying how evolutionary search depends on its initialization, we first construct an interpretable manually optimized program for each target model. Our goal is not to exhaustively optimize inference, but to establish a competitive starting point while keeping the resulting program interpretable and exposing distinct components that can later serve as mutation targets. We optimize three aspects of the inference pipeline: execution backend and batching, response generation, and image resolution. These choices define the manually optimized initialization used in the experiments of Section 6.3.

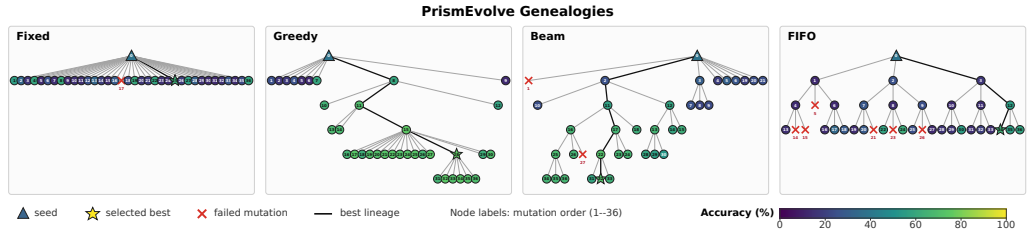
5.1 Inference Backend and Batching

We first select an execution configuration that supports efficient evaluation during evolutionary search. We compare Hugging Face Transformers [11] and vLLM [3] across batch sizes, keeping prompts and generation settings fixed. Runtime includes model initialization and inference over the full 128-example optimization set. As shown in Figure 2a, batching substantially reduces evaluation time. Transformers is fastest at batch size 64 but runs out of memory at 128, whereas vLLM supports batch size 128. Although vLLM's initialization overhead makes its total runtime slightly higher than Transformers at batch size 64, we use vLLM with batch size 128 thereafter because it supports the largest batch and provides a uniform execution backend for the controlled experiments.

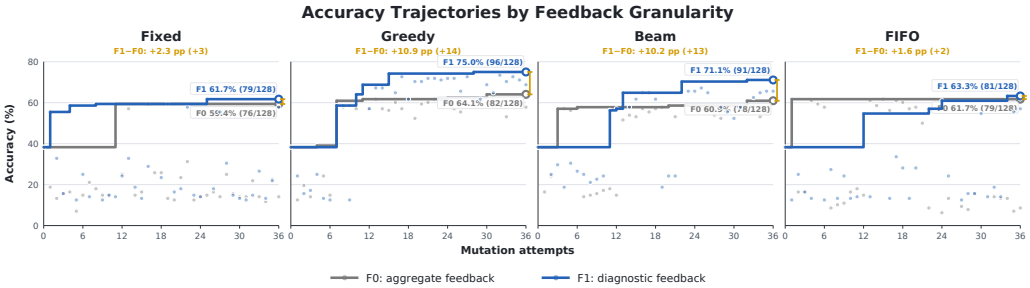
5.2 Response Structure and Generation Budget

We study four response configurations that combine two output protocols with optional type-specific guidance, which provides a task-specific procedure based on the detected CharXiv question type. *Direct* requests only the final answer with no explanation or reasoning, whereas *Structured* permits intermediate reasoning and requires the response to end with an explicit `Answer: <final answer>` line. Each protocol is evaluated both with and without type-specific guidance. For all four configurations, we sweep the maximum output length from 32 to 1024 tokens using greedy decoding and original-resolution images.

Observed trade-offs. Figure 2b shows different patterns for the two target models. For Instruct, the Direct configurations change little as the generation budget increases, whereas the Structured configurations benefit more from additional output length. Structured with guidance reaches the highest observed accuracy at 256 tokens, after which additional generation mainly increases runtime.



(a) Program genealogies under diagnostic feedback and accuracy-first selection. Labels denote mutation order, colors indicate accuracy, and bold edges trace the selected lineage.



(b) Accuracy trajectories under aggregate and diagnostic feedback. Dots show individual mutations, solid lines show best-so-far accuracy, and gold annotations report the final F1–F0 gap.

Figure 3: Search dynamics under alternative search and feedback designs. (a) Search policies allocate the mutation budget across different program genealogies. (b) Feedback granularity changes both individual candidate quality and best-so-far improvement.

Guidance also interacts with the generation budget, as Structured with guidance performs below Structured without guidance at 128 tokens but above it at 256 tokens.

For Thinking, accuracy generally increases with the generation budget, while the four configurations remain comparatively close over much of the sweep. Neither response structure nor type-specific guidance provides a consistent advantage across output lengths. At the largest tested budget, Direct without guidance attains the highest observed accuracy.

Selected configurations. Based on these observed accuracy–runtime trade-offs, we use Structured with type-specific guidance and a 256-token output budget for Instruct, and Direct without type-specific guidance and a 1024-token output budget for Thinking. The exact response templates, type-specific procedures, and extraction rules are provided in Appendix B.

5.3 Effect of Image Resolution

We next examine the effect of input image resolution while keeping the selected response configurations fixed. We vary the image long-edge resolution from 256 to 2048 pixels while preserving aspect ratio. Figure 2c shows that aggressive downsampling reduces accuracy, whereas increasing resolution beyond moderate values adds runtime without consistent accuracy gains. Based on this accuracy–runtime trade-off, we use 768 pixels for Instruct and 1536 pixels for Thinking.

6 What Makes Evolution Effective?

In this section, we examine which properties of the optimization loop determine whether evolutionary improvement accumulates effectively. We study the roles of candidate selection, evaluation feedback, initialization, and reuse of prior search experience under controlled search conditions.

6.1 Search Policy and Selection Objective

We analyze the effect of search policy using the same initial program, diagnostic feedback, no ancestry memory or archive inspiration, and a budget of 36 mutations. We compare Fixed, FIFO, Greedy, and Beam, and evaluate both accuracy-first and speed-first selection for Greedy and Beam, where candidate ranking determines which programs are expanded.

Inheritance and search trajectories. Figure 3a shows how the search policies allocate the same mutation budget across program genealogies. Fixed generates independent depth-one candidates from the initial program, whereas Greedy repeatedly expands a small number of promising descendants and reaches depth four. Beam distributes its mutations across multiple branches, while FIFO expands candidates in insertion order. These genealogies reveal different balances between exploring alternative branches and exploiting promising lineages.

Table 1: Search results on the 128-example optimization set. Correct denotes the number of correctly answered examples.

Policy	Objective	Correct (/128)	Time (s)
Fixed	–	79	23.73
FIFO	–	81	28.82
Greedy	Acc.-first	**96**	24.86
Beam	Acc.-first	91	38.84
Greedy	Speed-first	87	**20.78**
Beam	Speed-first	31	23.38

The differences in search dynamics are reflected in Table 1. Compared with Fixed, Greedy and Beam answer 17 and 12 more questions correctly, respectively, whereas FIFO gains only two correct answers despite allowing modifications to accumulate through inheritance. Under the same 36-mutation budget, Greedy achieves the highest accuracy while concentrating more of its search on promising descendants. Together, these results indicate that inheritance alone is insufficient; its benefit depends on how the search policy selects which descendants to expand and propagate.

Selection objective. We next vary the criterion used to rank candidates while keeping the mutation prompt unchanged. Accuracy-first selection yields higher final accuracy for both Greedy and Beam, whereas speed-first selection favors lower-runtime candidates and produces different search outcomes (Table 1). Because the mutation instructions are identical across objectives, the divergence comes from selection rather than generation. The ranking objective therefore shapes the entire evolutionary trajectory by determining which trade-offs are repeatedly inherited and amplified across generations.

6.2 Feedback Granularity and Search Dynamics

We next examine whether richer evaluation feedback is useful on its own or mainly when the search policy can exploit it. Aggregate feedback contains only program-level measurements, whereas diagnostic feedback additionally exposes instance-level outcomes.

Figure 3b shows that diagnostic feedback improves all four policies, with much larger gains for Greedy and Beam than for Fixed and FIFO. The individual mutation scores and best-so-far curves show that Greedy and Beam continue to discover higher-scoring candidates under diagnostic feedback, producing larger cumulative gains over the course of search.

This pattern is consistent with their search policies, which repeatedly select strong candidates for further mutation, whereas Fixed always returns to the initialization and FIFO expands candidates in insertion order. These results suggest that fine-grained feedback is most useful when the search policy can selectively carry promising improvements forward.

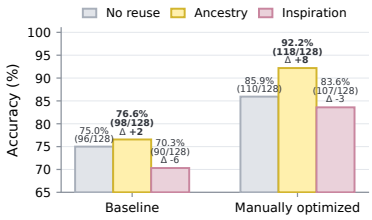
6.3 Initialization and Evolvability

We next examine how the starting program affects the outcome of subsequent evolution. We use *evolvability* to describe a starting program's capacity to yield strong descendants under a fixed search configuration and mutation budget. We therefore compare the endpoints reached from different starting programs under matched search conditions, while using improvement magnitude and the frequency of improving mutations as complementary indicators. We hold the remaining search settings fixed and allocate 36 mutation attempts to every condition.

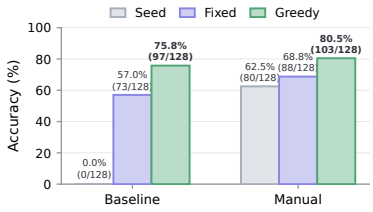
Table 2 shows that higher initial accuracy does not necessarily lead to a stronger endpoint after further evolution. Under both Greedy and Beam, the previously evolved program starts above the

Table 2: Effect of the starting program on the 128-example optimization set. Initial, final, and gain are reported as numbers of correct answers. Improving attempts produce a program that scores above its respective starting program; failed or duplicate attempts are counted as non-improving.

Policy	Starting program	Initial	Final	Gain	Improving attempts
Greedy	Baseline	49	96	+47	29/36
	Manually optimized	88	**110**	+22	20/36
	Previously evolved	**96**	100	+4	5/36
Beam	Baseline	49	91	+42	22/36
	Manually optimized	88	**101**	+13	18/36
	Previously evolved	**91**	94	+3	4/36



(a) Search-experience reuse from baseline and manually optimized initializations. Labels show accuracy and correct answers out of 128.



(b) Transfer to Qwen3-VL-2B-Thinking under Fixed and Greedy. Labels show accuracy and correct answers out of 128.

Figure 4: Analyses of search-experience reuse and transfer of the evolutionary procedure.

manually optimized program, yet evolution from the manual start reaches a higher endpoint. This reversal shows that a program that is stronger initially can still be a worse starting point for continued evolution under the same search policy and mutation budget.

One possible explanation is that the starting programs expose different opportunities for subsequent modification. The manually optimized program separates question classification, type-specific guidance, image preprocessing, and answer parsing into explicit components, whereas the previously evolved program relies more heavily on shared global mechanisms. This difference is also reflected in the frequency of improving mutations, which is substantially higher from the manual start. Although this does not establish a causal effect of program structure, it is consistent with the interpretation that explicit, behavior-specific components provide more actionable mutation targets for continued search. Complete best-so-far curves and a structural comparison are provided in Appendix C.

6.4 Effects of Search-Experience Reuse

We examine whether providing additional information from prior evolution helps subsequent mutations. We compare no additional reuse with ancestry memory, which summarizes the selected parent's lineage, and archive inspiration, which provides a randomly sampled evaluated program from elsewhere in the archive. We run these comparisons from both the baseline and manually optimized initializations while keeping the remaining search configuration fixed.

Figure 4a summarizes the final best-so-far scores. Ancestry memory reaches a higher observed endpoint under both initializations, with the larger gain occurring from the manually optimized start (+8 correct answers). In contrast, random archive inspiration does not improve the endpoint in either setting.

These results should be interpreted as evidence about the particular reuse mechanisms studied here rather than as a general conclusion about prior evolutionary experience. They suggest that additional context is not uniformly helpful, and that its usefulness may depend on how closely the reused information is tied to the current lineage and how it is presented to the mutation model. Complete best-so-far trajectories for all reuse conditions are provided in Appendix D.

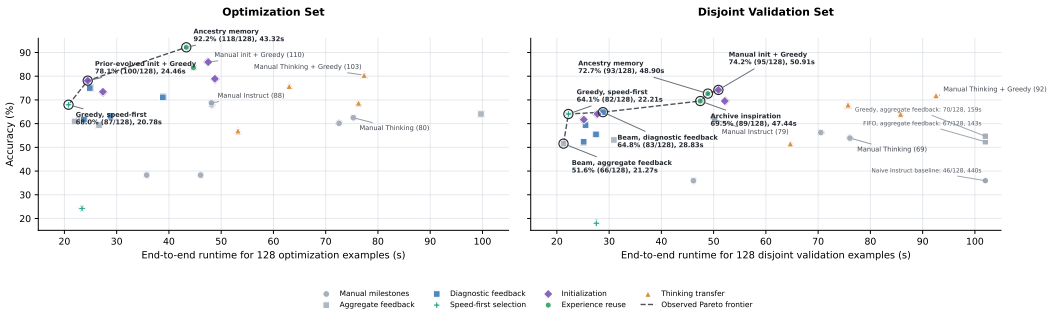


Figure 5: Accuracy–runtime trade-offs for representative programs reported in this paper. The left panel shows the optimization set, and the right panel evaluates the same frozen programs on a disjoint validation set that was not used to generate mutation feedback. Dashed lines show the observed Pareto frontiers. Labels report accuracy, correct answers out of 128, and end-to-end runtime, including model initialization and inference.

6.5 Transfer of the Evolutionary Procedure

We next test whether the evolutionary procedure identified on Qwen3-VL-2B-Instruct transfers to a different target model. We keep the mutation model, diagnostic feedback, accuracy-first selection, and 36-attempt budget fixed, and apply the same procedure to Qwen3-VL-2B-Thinking. Fixed repeatedly mutates the initialization, whereas Greedy repeatedly selects and expands the best program found so far. Figure 4b shows two consistent patterns. Greedy reaches higher endpoints than Fixed under both initializations, and the manually optimized start reaches higher endpoints than the baseline under both search policies.

These results reproduce the main search-policy pattern observed on Qwen3-VL-2B-Instruct and show that the same evolutionary procedure remains effective after changing the target VLM. The zero score of the baseline seed reflects a mismatch between the baseline inference protocol and the Thinking model rather than a limitation of the model itself. Complete best-so-far trajectories are provided in Appendix E.

6.6 Generalization and Final Program Selection

We evaluate representative programs from the preceding experiments on both the 128-example optimization set and a disjoint 128-example validation set containing different charts. The validation set is not used to generate mutation feedback, but is used when assigning the final submission roles. We report one point per distinct program, including representative endpoints from the preceding experiments.

Relative to the naive Instruct baseline, the manually optimized and evolved programs improve both accuracy and runtime substantially. The runtime gains come primarily from batching and the vLLM backend, while response design, task-specific guidance, image-resolution tuning, and subsequent evolution contribute to the accuracy improvements.

As shown in Figure 5, several improvements discovered on the optimization set remain visible on the validation set, although the relative ordering of programs changes. Evolving the manually optimized Instruct program increases the number of correct answers from 88 to 110 on the optimization set and from 79 to 95 on validation. The corresponding Thinking program improves from 80 to 103 and from 69 to 92, respectively. Thus, the gains produced by evolution are not limited to the examples used as mutation feedback.

The relative ordering is not preserved exactly. Ancestry memory obtains the highest number of correct answers on the optimization set, with 118, whereas the manually initialized Greedy program without additional experience reuse obtains the highest number on the validation set, with 95. These differences motivate considering optimization accuracy, validation accuracy, and runtime separately when assigning the final submission roles.

Table 3: Final submission programs and the evolutionary configurations used to obtain them. Correct is the number of correct answers out of 128, and time is end-to-end runtime in seconds. Repeated rows indicate that the same program serves multiple submission roles.

		Evolution configuration				Optimization set		Validation set	
Submission	Model	Init.	Policy	Objective	Experience	Correct	Time (s)	Correct	Time (s)
Manual Instruct	Instruct	Manual	–	–	–	88	48.15	79	50.19
Manual Thinking	Thinking	Manual	–	–	–	80	75.29	69	76.08
Evolved Instruct	Instruct	Manual	Greedy	Acc.-first	None	110	47.52	**95**	50.91
Evolved Thinking	Thinking	Manual	Greedy	Acc.-first	None	103	77.35	92	92.56
Best accuracy	Instruct	Manual	Greedy	Acc.-first	None	110	47.52	**95**	50.91
Best speed	Instruct	Baseline	Greedy	Speed-first	None	87	**20.78**	82	**22.21**
Best overall	Instruct	Manual	Greedy	Acc.-first	Ancestry	**118**	43.32	93	48.90

As summarized in Table 3, the evolved submissions are obtained with Greedy search and accuracy-first selection from the manually optimized initializations, without additional experience reuse. These programs reach 95 and 92 correct answers on the validation set for Instruct and Thinking, respectively. The Instruct program is also used for the best-accuracy submission because it achieves the highest validation accuracy among the evaluated programs.

The best-speed submission is obtained with Greedy search and speed-first selection from the baseline initialization, again without additional experience reuse. It is the fastest point on the optimization-set Pareto frontier and retains 82 correct answers on the validation set.

For the best-overall submission, we use Greedy search with accuracy-first selection from the manually optimized initialization together with ancestry memory. This program reaches 118 correct answers on the optimization set and 93 on the validation set, while running slightly faster than the best-accuracy program. Because we do not define a scalar objective that combines accuracy and runtime, the best-overall designation is a heuristic selection rather than the optimum of a formal combined score.

7 Conclusion and Limitations

We studied what makes LLM-driven program evolution effective by separating key components of the optimization loop in PRISMEVOLVE. Across controlled experiments on VLM inference optimization, we find that useful mutations do not accumulate automatically. Improvements are most effectively compounded when the search policy repeatedly selects and propagates promising descendants, and richer evaluation feedback provides its largest benefit when the search procedure can act on that information. We also find that initial accuracy alone does not determine evolvability. A structured manually optimized program can yield substantially stronger descendants than a higher-accuracy previously evolved program under the same mutation budget. Finally, prior evolutionary experience is not uniformly useful. Lineage-specific ancestry memory reaches a higher observed endpoint in our setting, whereas randomly sampled archive inspiration does not. Together, these results show that the design of the optimization loop determines which improvements are retained, which feedback is made actionable, and whether program structure remains amenable to further modification.

Our study has several limitations. First, the experiments focus on VLM inference optimization for CharXiv and two closely related Qwen3-VL-2B models, so the extent to which the observed patterns generalize to other program-optimization tasks, model families, or larger search spaces remains unknown. Second, each evolutionary condition uses a limited 36-mutation budget and a fixed mutation model, and we do not study how the conclusions change with substantially larger budgets or different mutation models. Third, the optimization and validation splits contain 128 examples each, and although the validation set is disjoint and never used for mutation feedback, broader evaluation would provide stronger evidence of generalization. Finally, some of our explanations are mechanistic interpretations rather than causal conclusions. In particular, the association between modular program structure and evolvability, and the benefit of lineage-specific ancestry memory, are supported by the observed comparisons but are not independently isolated. Future work could test these hypotheses across broader tasks and models and develop more principled mechanisms for selecting, representing, and reusing evolutionary experience.

References

[1] Shuai Bai, Yuxuan Cai, Ruizhe Chen, Keqin Chen, Xionghui Chen, Zesen Cheng, Lianghao Deng, Wei Ding, Chang Gao, Chunjiang Ge, Wenbin Ge, Zhifang Guo, Qidong Huang, Jie Huang, Fei Huang, Binyuan Hui, Shutong Jiang, Zhaohai Li, Mingsheng Li, Mei Li, Kaixin Li, Zicheng Lin, Junyang Lin, Xuejing Liu, Jiawei Liu, Chenglong Liu, Yang Liu, Dayiheng Liu, Shixuan Liu, Dunjie Lu, Ruilin Luo, Chenxu Lv, Rui Men, Lingchen Meng, Xuancheng Ren, Xingzhang Ren, Sibo Song, Yuchong Sun, Jun Tang, Jianhong Tu, Jianqiang Wan, Peng Wang, Pengfei Wang, Qiuyue Wang, Yuxuan Wang, Tianbao Xie, Yiheng Xu, Haiyang Xu, Jin Xu, Zhibo Yang, Mingkun Yang, Jianxin Yang, An Yang, Bowen Yu, Fei Zhang, Hang Zhang, Xi Zhang, Bo Zheng, Humen Zhong, Jingren Zhou, Fan Zhou, Jing Zhou, Yuanzhi Zhu, and Ke Zhu. Qwen3-VL technical report. *arXiv preprint arXiv:2511.21631*, 2025. doi: 10.48550/arXiv.2511.21631. URL https://arxiv.org/abs/2511.21631.

[2] Dongzhi Jiang, Renrui Zhang, Ziyu Guo, Yanwei Li, Yu Qi, Xinyan Chen, Liuhui Wang, Jianhan Jin, Claire Guo, Shen Yan, Bo Zhang, Chaoyou Fu, Peng Gao, and Hongsheng Li. MME-CoT: Benchmarking chain-of-thought in large multimodal models for reasoning quality, robustness, and efficiency. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pages 27793–27830. PMLR, 2025. URL https://proceedings.mlr.press/v267/jiang25n.html.

[3] 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 *Proceedings of the 29th Symposium on Operating Systems Principles*, 2023. doi: 10.1145/3600006.3613165. URL https://doi.org/10.1145/3600006.3613165.

[4] Robert Tjarko Lange, Yuki Imajuku, and Edoardo Cetin. ShinkaEvolve: Towards open-ended and sample-efficient program evolution. *arXiv preprint arXiv:2509.19349*, 2025. doi: 10.48550/arXiv.2509.19349. URL https://arxiv.org/abs/2509.19349.

[5] Fei Liu, Xialiang Tong, Mingxuan Yuan, Xi Lin, Fu Luo, Zhenkun Wang, Zhichao Lu, and Qingfu Zhang. Evolution of heuristics: Towards efficient automatic algorithm design using large language model. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pages 32201–32223. PMLR, 2024. URL https://proceedings.mlr.press/v235/liu24bs.html.

[6] Fei Liu, Rui Zhang, Zhuoliang Xie, Rui Sun, Kai Li, Xi Lin, Zhenkun Wang, Zhichao Lu, and Qingfu Zhang. LLM4AD: A platform for algorithm design with large language model. *arXiv preprint arXiv:2412.17287*, 2024. doi: 10.48550/arXiv.2412.17287. URL https://arxiv.org/abs/2412.17287.

[7] 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, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, and Matej Balog. AlphaEvolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025. doi: 10.48550/arXiv.2506.13131. URL https://arxiv.org/abs/2506.13131.

[8] 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(7995):468–475, 2024. doi: 10.1038/s41586-023-06924-6. URL https://www.nature.com/articles/s41586-023-06924-6.

[9] Asankhaya Sharma. OpenEvolve: An open-source evolutionary coding agent. GitHub repository, 2025. URL https://github.com/algorithmicsuperintelligence/openevolve.

[10] 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*, volume 37, 2024. doi: 10.52202/079017-3609.

[11] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Remi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45. Association for Computational Linguistics, 2020. doi: 10.18653/v1/2020.emnlp-demos.6. URL `https://aclanthology.org/2020.emnlp-demos.6/`.

A Experimental Configuration and Reproducibility

Data and scoring. All search decisions use the first 128 descriptive questions produced by the CharXiv validation annotations in their original order. The disjoint validation analysis uses a non-overlapping block of 128 questions beginning at offset 256 and is never supplied as mutation feedback. We score a prediction by case-insensitive exact match after stripping leading and trailing whitespace. Failed examples count as incorrect.

Table 4: Common configuration for the evolutionary experiments. Parameters in the lower part of the table are varied only in the named ablations.

Component	Configuration
Target models	Qwen3-VL-2B-Instruct; Qwen3-VL-2B-Thinking
Mutation model	Gemini 3.6 Flash, medium reasoning
Mutation decoding	provider-default sampling; one candidate; 16,384 output-token cap
Mutation format	JSON search-and-replace edits
Optimization data	128 CharXiv descriptive questions, offset 0
Validation data	128 disjoint descriptive questions, offset 256
Candidate decoding	greedy ($T = 0$)
Search budget	36 mutation attempts
Expansion size	3 children per selected parent (12 expansions)
Evaluation seed	42
Accuracy objective	maximize correct count; runtime breaks ties
Speed objective	minimize end-to-end runtime; accuracy breaks ties
Runtime field	import + initialization + 128-example evaluation
Default parent policy	Greedy
Default feedback	diagnostic, per-example feedback
Default experience reuse	none
Beam width	3
Archive-inspiration count	1 when enabled; otherwise 0
Archive sampling seed	42

Mutation prompt construction. Each request contains a fixed system instruction and a structured JSON payload. The base system instruction used in the reported experiments is shown below (line wrapping is changed only for typesetting):

```
Further improve the provided parent program to increase
answer accuracy and reduce evaluation time relative to
this parent.

Consider a broad range of strategies across the inference
pipeline, and use the parent's current code and measured
results to identify opportunities for further improvement.

When reference programs are provided, consider their code
and measured results as inspiration for exploring diverse
ways to further improve the current parent program.

Return a concrete modification to the current parent.
Do not describe an existing feature as a new change or
return search/replace edits with identical content.

Do not change the designated model or model variant.
Use greedy decoding.
Preserve the vlm_inference interface.

Modify only the parent program.
Reference programs, when supplied, are read-only.
Treat the supplied evaluator and question-building sources
as read-only.

Return JSON matching the supplied schema: summary and edits.
Each search must match exactly once, including whitespace
and newlines.
```

The user message begins with `INPUT_JSON:` followed by the payload below. Fields in brackets are included only in the corresponding experimental condition.

```
{
  "model": <target model>,
  "parent_code": <complete current parent source>,
  "runtime_environment": <GPU and package metadata>,
  "interface_contract": <vlm_inference contract>,
  "evaluation_context": [
    <evaluator source>, <question-builder source>
  ],
  "feedback": {
    "aggregate": <score, runtime, and execution status>,
    ["per_question": <prediction, reference, correctness>],
    ["categories": <accuracy grouped by question type>]
  },
  ["inspirations": <reference code and matched feedback>],
  ["ancestry_history": <ordered edits and measured deltas>]
}
```

The response is constrained to JSON with a short mutation summary and a list of exact `search/replace` edits. The controller verifies that each search string occurs exactly once, applies the edits only to the parent, validates the resulting Python program, and then evaluates it. Thus feedback, ancestry, and archive inspiration change the request context without changing the required output format.

Controlled comparisons. Fixed always mutates the initialization; FIFO expands eligible programs in creation order; Greedy expands the highest-ranked program in the full archive; and Beam retains the three highest-ranked incumbents after completing each round. Aggregate feedback contains execution status, correct count, accuracy, and runtime. Diagnostic feedback additionally contains each question's type, prediction, reference answer, and correctness. Ancestry memory supplies the ordered edits and measured score changes on the selected parent's lineage. Archive inspiration supplies one eligible program sampled from outside that lineage. Except for the component named in each comparison, the common configuration in Table 4 is unchanged.

Table 5: Factors varied in the analysis. "Manual" denotes the model-specific manually optimized program.

Analysis	Varied factor	Values
Search policy	parent selection	Fixed, FIFO, Greedy, Beam
Objective	ranking	accuracy-first, speed-first
Feedback	evaluator context	aggregate, diagnostic
Initialization	starting program	baseline, manual, prior evolved
Experience reuse	mutation context	none, ancestry, archive inspiration
Thinking transfer	policy and seed	Fixed/Greedy; baseline/manual

Inference and runtime measurement. Unless a candidate mutation changes it, vLLM uses FP16, a maximum context of 8,192 tokens, GPU-memory utilization 0.90, eager execution, and batches of 128. The submitted manual Instruct program uses a 768-pixel image long edge and a 256-token generation cap; manual Thinking uses a 1,536-pixel long edge and a 1,024-token cap. The submitted evolved Instruct and Thinking programs use 1,280/256 and 1,536/1,280 pixels/tokens, respectively. Runtime is measured with `time.perf_counter` and includes candidate import, model and processor initialization, and the complete 128-example inference loop. It excludes evaluator setup, dataset construction, metadata collection, and summary-file writing. We perform no warm-up run.

Hardware and software. All reported local measurements use one NVIDIA GeForce RTX 5090 (32,607 MiB), driver 595.84, CUDA 13.2, and Python 3.12.3. The main inference stack is vLLM 0.29.0, Transformers 5.17.0, PyTorch 2.13.0+cu132, Torchvision 0.28.0+cu132, and qwen-vl-utils 0.0.14.

B Manual Response Protocols and Task Guidance

This appendix provides the exact response templates, type-specific guidance, and answer-extraction rules used in the response-generation sweep of Section 5.2. Across these experiments, we vary the response protocol, optional type-specific guidance, answer extraction, and maximum generation length while keeping the remaining inference settings fixed.

The Direct condition uses the following system instruction:

```
You are an expert chart-reading assistant.
Carefully inspect the requested plot or subplot and follow
every answer-format rule in the user's question.
Return only the exact final answer, with no explanation,
reasoning, preamble, label, or trailing commentary.
Copy visible textual labels exactly as written in the chart.
When the requested information is absent under the user's
rules, return exactly "Not Applicable".
```

The Structured condition instead permits intermediate reasoning and requests an explicit final-answer field:

```
You are an expert chart-reading assistant.
Carefully inspect the requested plot or subplot and follow
every answer-format rule in the user's question.
Analyze the chart carefully and solve the question. You may
reason step by step.
End your response with exactly one final line in this form:
Answer: <final answer>
Do not write anything after the Answer line.
Copy visible textual labels exactly as written in the chart.
When the requested information is absent under the user's
rules, return exactly "Not Applicable" as the final answer.
```

Without type-specific guidance, the original CharXiv question is used unchanged. With type-specific guidance, the system prompt additionally states, "Use the supplied task-specific procedure to avoid confusing chart components." A deterministic string matcher then assigns one of 19 question types and constructs the user text as follows:

```
[Original CharXiv question]
<original question>

[Task-specific procedure]
<procedure for the matched question type>
```

The complete task-specific procedures used in the experiment are:

```
[title]
First locate the requested plot. Read only a title explicitly
written for that plot. Do not use a subplot letter, axis label,
legend text, or surrounding caption as its title.

[x_axis_label]
First locate the requested plot and inspect its bottom x-axis. Use
an explicitly written axis name, including a shared x-axis label. Do
not return a tick value, scale annotation, unit alone, title, or
legend label.

[y_axis_label]
First locate the requested plot and inspect its left y-axis. Use an
explicitly written axis name, including a shared y-axis label. Do
not return a tick value, scale annotation, unit alone, title, or
legend label.

[leftmost_x_tick]
Use the bottom x-axis of the requested plot, including a shared
axis. Return its spatially leftmost explicitly printed tick. Do not
infer a data minimum or apply a separately written unit or scale.

[rightmost_x_tick]
Use the bottom x-axis of the requested plot, including a shared
axis. Return its spatially rightmost explicitly printed tick. Do not
infer a data maximum or apply a separately written unit or scale.
```

```
[lowest_y_tick]
Use the left y-axis of the requested plot, including a shared axis.
Return its spatially lowest explicitly printed tick. Do not infer
the minimum plotted data value or apply a separate scale.

[highest_y_tick]
Use the left y-axis of the requested plot, including a shared axis.
Return its spatially highest explicitly printed tick. Do not infer
the maximum plotted data value or apply a separate scale.

[x_tick_difference]
Read multiple consecutive numerical ticks on the bottom x-axis.
Verify that every adjacent difference is constant, then report that
difference. Do not apply separately written units or scales.

[y_tick_difference]
Read multiple consecutive numerical ticks on the left y-axis. Verify
that every adjacent difference is constant, then report that
difference. Do not apply separately written units or scales.

[line_count]
Count plotted data lines in the requested plot only. Exclude axes,
borders, grid lines, tick marks, error bars, colorbar edges, and
vertical or horizontal auxiliary reference lines.

[line_intersection]
Check intersections between plotted data lines in the requested plot
only. Ignore contacts with axes, borders, grid lines, tick marks,
error bars, and auxiliary reference lines.

[legend_count]
Identify the discrete legend relevant to the requested plot, even
when it lies outside the axes. Count its labeled entries only. Do
not count continuous colorbar ticks or unrelated subplot legends.

[legend_labels]
Identify the discrete legend relevant to the requested plot, even
when it lies outside the axes. Copy its visible entry text exactly,
ordered top-to-bottom and then left-to-right. Do not include
continuous colorbar ticks or unrelated subplot legends.

[colorbar_range]
Locate a separate continuous gradient scale with numerical tick
labels that is relevant to the requested plot. Do not treat a plot
axis, plotted data, or discrete legend as a colorbar. Subtract its
minimum printed tick from its maximum printed tick and omit a
percentage sign from the final answer.

[colorbar_max]
Locate a separate continuous gradient scale with numerical tick
labels that is relevant to the requested plot. Do not use plot-axis
ticks, plotted data, or discrete legend entries. Return its maximum
printed tick and omit a percentage sign from the final answer.

[trend]
Trace the data in the requested plot from left to right and describe
its overall direction in a few words. Do not let small local
fluctuations dominate unless they change the overall pattern.

[tick_count]
Count explicitly printed tick labels across every axis, including
shared axes. Count each visible label once. Exclude unlabeled tick
marks, axis names, titles, legend entries, annotations, and colorbar
ticks.

[subplot_layout]
Identify distinct data panels, then count their rows and columns. Do
not treat legends, colorbars, tables, inset annotations, or subplot
letters as additional panels. Format the result as rows by columns.

[subplot_count]
Count distinct data panels in the entire figure. Do not count
legends, colorbars, tables, inset annotations, or subplot letters as
additional panels. A figure with one data panel has one subplot.
```

For Direct responses, leading and trailing whitespace is stripped and the remaining text is returned as the answer. For Structured responses, the final line matching `Answer: <final answer>` is

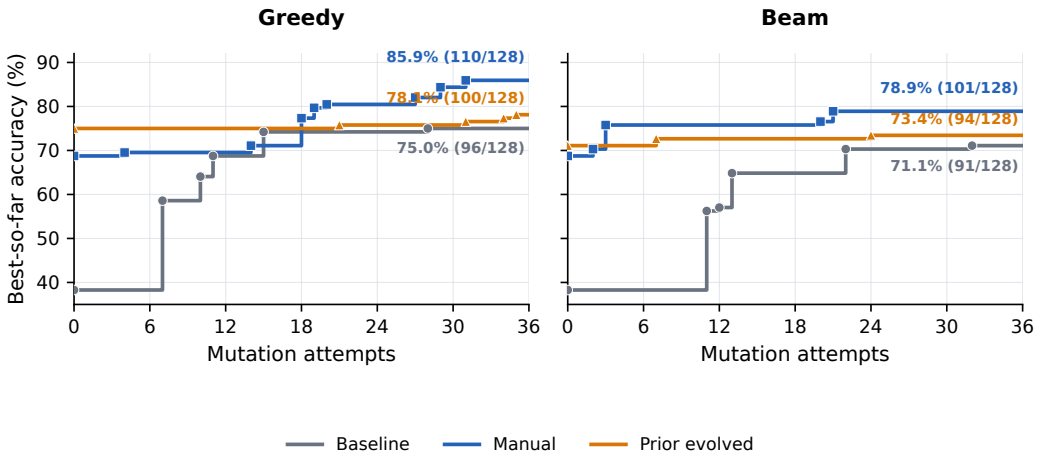


Figure 6: Best-so-far optimization accuracy from baseline, manually optimized, and previously evolved programs. Although the previously evolved programs begin above the manual program, they improve less and converge to lower endpoints under both Greedy and Beam.

Algorithm 1 Execution structure of the two Greedy starting programs. The manual start exposes task routing, visual preprocessing, and answer parsing as separate mutation targets, whereas the previously evolved start shares most behavior across question types.

(a) Manually optimized start

Require: image I, question q

$c \leftarrow \textsc{ClassifyQuestion}(q, \mathcal{T}_{19})$
$g \leftarrow \textsc{TypeSpecificGuidance}(c)$
$\widetilde{I} \leftarrow \textsc{Resize}(I, r)$
$p \leftarrow \textsc{ComposePrompt}(q, g)$
$y \leftarrow \text{VLM}(\widetilde{I}, p)$
return $\textsc{ParseAnswer}(y)$

Separate mutation targets: classifier, per-type guidance, resolution, and parser.

(b) Previously evolved start

Require: image I, question q

$p \leftarrow \textsc{GlobalPrompt}$
$g \leftarrow \textsc{BroadKeywordRule}(q)$
$p' \leftarrow \textsc{ComposePrompt}(p, q, g)$
$y \leftarrow \text{VLM}(I, p')$
return $\textsc{SharedCleaner}(y)$

Shared mutation targets: global prompt, broad routing, and response cleaner.

extracted and only the captured answer is returned. For Qwen3-VL-2B-Thinking, both extractors first discard the reasoning trace through the final `</think>` marker.

The generation-length sweep changes only the greedy-decoding output cap, using 32, 64, 128, 256, 512, and 1024 tokens.

C Additional Initialization Analysis

Figure 6 provides the complete best-so-far trajectories for the initialization experiments in Section 6.3. Under the matched 36-mutation budget, the manually optimized start reaches the highest endpoint under both Greedy and Beam. Although the previously evolved starts begin with higher accuracy, they improve less and plateau at lower endpoints. These trajectories further illustrate that initial accuracy alone does not explain the differences in subsequent improvement.

Algorithm 1 summarizes the execution structure of the manually optimized and previously evolved Greedy starts. The manual program exposes question classification, type-specific guidance, image preprocessing, and answer parsing as separate components. In contrast, the previously evolved program shares more of its behavior through a global prompt, broad routing rules, and a common response cleaner. This comparison provides the structural context for the mutation-opportunity analysis discussed in Section 6.3.

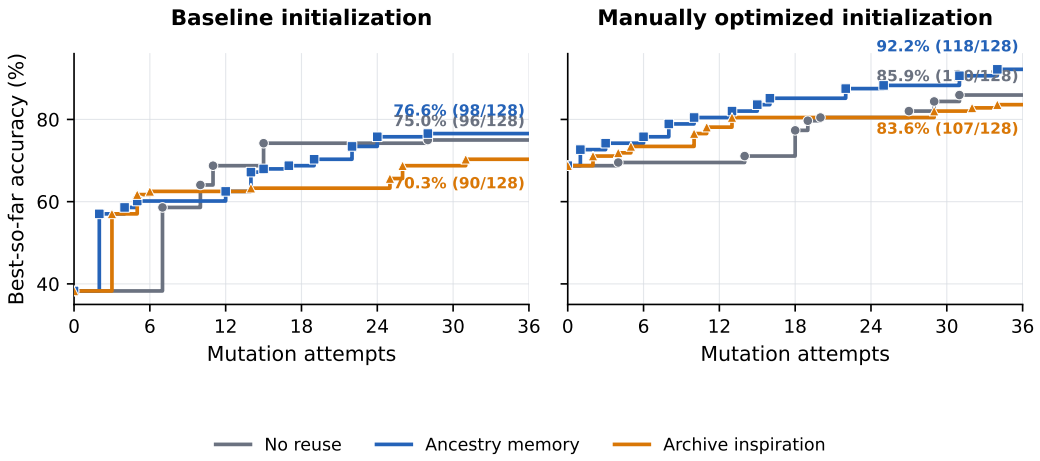


Figure 7: Best-so-far optimization accuracy under no reuse, ancestry memory, and archive inspiration for the baseline and manually optimized initializations. Endpoint labels report both accuracy and the number of correct answers out of 128 optimization examples.

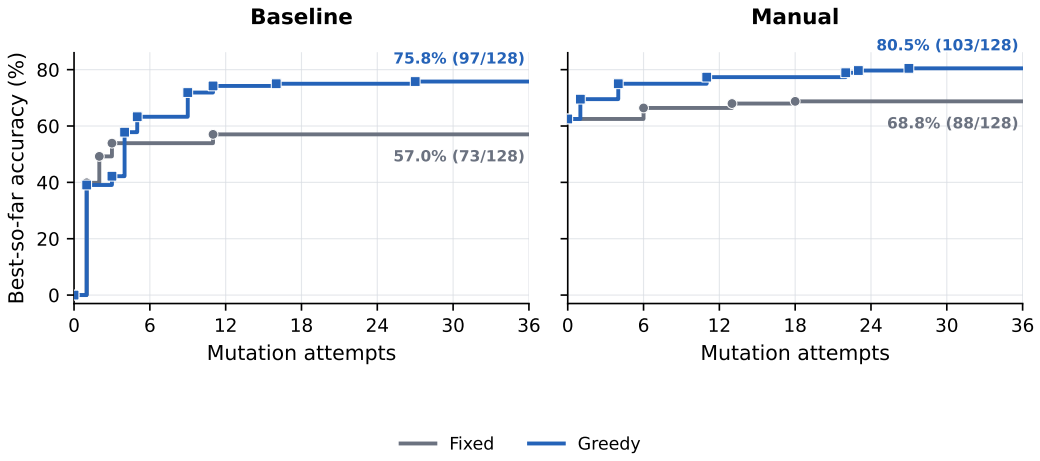


Figure 8: Best-so-far optimization accuracy for Fixed and Greedy search on Qwen3-VL-2B-Thinking from the baseline and manually optimized initializations. Endpoint labels report both accuracy and the number of correct answers out of 128 optimization examples.

D Additional Analysis of Search-Experience Reuse

Figure 7 shows the complete best-so-far trajectories for the search-experience reuse experiments. Under both initializations, ancestry memory reaches a higher observed endpoint than no additional reuse, while archive inspiration does not improve the endpoint in these runs. The larger gain from ancestry memory under the manually optimized initialization is suggestive, but we do not isolate this interaction further.

E Additional Analysis of Thinking Transfer

Figure 8 shows the complete best-so-far trajectories for the transfer experiments on Qwen3-VL-2B-Thinking. Greedy reaches a higher observed endpoint than Fixed from both the baseline and the manually optimized Thinking initialization, consistent with the comparison reported in Section 6.5.