Agentic Superoptimization of Bioimaging Analysis Workflows

Xuefei (Julie) Wang* Caltech
Jonathan Chen Cornell
Alexander R. Farhang Caltech
Sophia Stiles Caltech
Kai A. Horstmann Cornell
Atharva Sehgal UT Austin
Jonathan Light Rensselaer Polytechnic Institute
David Van Valen Caltech
Yisong Yue Caltech
Jennifer J. Sun Cornell

Abstract

Data-driven scientific discovery relies on complex computational workflows to process large, high-dimensional experimental datasets. However, a fundamental bottleneck exists: adapting carefully engineered computational tools to bespoke datasets studied by individual labs demands substantial manual tuning and custom code development, consuming weeks or months of expert time and slowing scientific progress. To address this bottleneck, we introduce agentic superoptimization, a new paradigm for leveraging generative AI to autonomously write customized code that can surpass human-expert-engineered solutions. We study a proof-of-concept agentic framework for superoptimizing data preparation functions directly in real-world, production-level scientific workflows, without requiring additional annotations or training. We validate our approach on challenging biology and medical imaging tasks, consistently outperforming expert baselines. Notably, our agent-generated code achieved the first-ever successful deployment into a production-level scientific pipeline. Our work lays the foundation towards human-AI agent collaborative discovery in complex, real-world environments.

1 Introduction

The past decade has seen an unprecedented acceleration in data-driven scientific discovery, with researchers leveraging increasingly sophisticated computational workflows to tackle the ever-expanding volume and complexity of experimental data [31]. These advances, however, have revealed a fundamental bottleneck in adapting these powerful systems to the specific requirements of individual researchers. In almost all real-world scientific inquiry, the application of carefully engineered computational tools to new datasets (even those within the same domain) commonly demands substantial manual tuning and custom code development across all stages of the scientific workflow [30].

For instance, modern cell segmentation tools often fail on new images because of significant variability in acquisition conditions between labs (e.g., lighting, noise, resolution) [14]. Achieving useful results requires extensive manual effort in data preparation, hyperparameter tuning, and postprocessing. These efforts can range from heuristic function development based on manual inspection of images to training custom denoising neural networks, consuming months of invaluable domain expert time. This

*Correspondence: xwang3@caltech.edu
Code available at: `https://github.com/xuefei-wang/agent-super-opt`

COLM 2025 Workshop LM4Sci

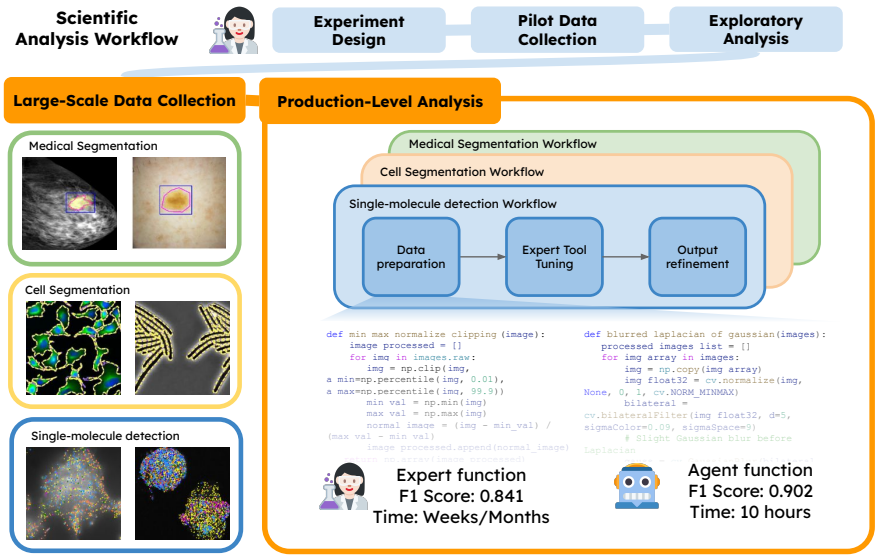


Figure 1: **Overview of Scientific Analysis Workflow with Agentic Superoptimization**. We consider three case settings in biomedical imaging analysis that span the multiple length scales, from single molecule to macroscopic (Section 4).

inefficiency slows the pace of scientific progress, limits the reproducibility and utility of cutting-edge computational tools.

We propose agentic superoptimization of scientific analysis workflows, a new paradigm that transforms how complex scientific analyses are developed and deployed. Within this framework, superoptimization extends beyond its traditional focus on generating maximally optimized (e.g., fastest or smallest) code, to achieving scientific analysis outcomes that surpass human-expert-engineered solutions. This unique combination of advanced agentic capabilities with the objective of surpassing expert-level performance in integrated real-world production-level scientific workflows distinguishes our work from existing agentic systems that typically focus on exploratory data analysis [22], task automation [4], or program correctness [1].

We study a proof-of-concept agent framework to evaluate the effectiveness of agentic superoptimization, demonstrating that these agents can indeed develop solutions that outperform expert-designed workflows. Our work provides significant evidence that common agent design choices may not be beneficial for complex, real-world tasks. A key insight is that the primary bottleneck in these scientific applications is the evaluation process—in terms of both compute and time—rather than the LLM calls themselves. This suggests there is less incentive for domain experts to use faster but less performant LLMs. Furthermore, we find that function banks, a frequently used module, can surprisingly limit solution diversity and prove ineffective for scientific discovery. Understanding and navigating these design choices presents unique and demanding challenges, pushing the boundaries of what LLM agents can achieve in real-world environments.

We demonstrate the real-world impact of agentic superoptimization by deploying the first agent-generated functions into a production pipeline, which automates painstaking manual code adaptations in science (Figure 4a). Looking beyond the biomedical imaging in this study, future applications in data standardization, agent tool use, and human-AI collaboration can further accelerate scientific discovery.

2 Problem Statement

We address the problem of superoptimizing scientific analysis workflows, as depicted in Figure 1. Given a production-level analysis tool for biomedical imaging, the objective is to develop and configure components of workflows to maximize performance on a targeted benchmark. In particular, one needs to write programs to instantiate such workflows, resulting in a combinatorially (or infinitely) large search space.

Superoptimization. The term superoptimization comes from the program synthesis community [25]—given a reference function π_{ref}, the goal is to find a program π in a programming language

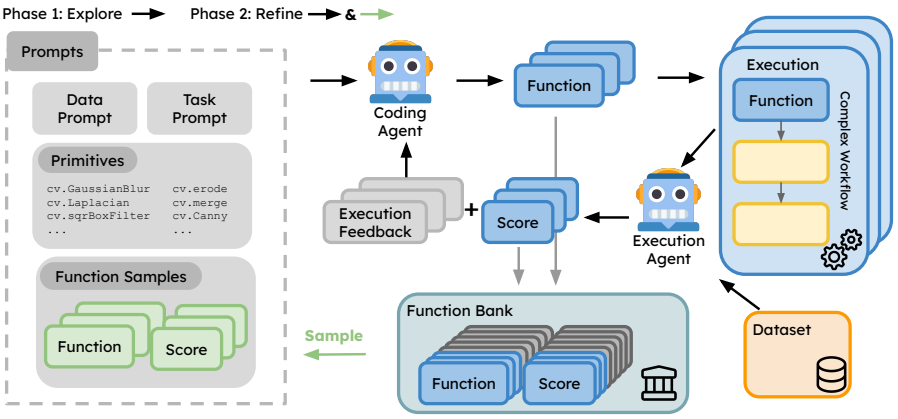


Figure 2: **Agent Framework for Superoptimization**. The key components include a dual-agent generation-execution feedback loop and a function bank.

$\mathcal{L}$ that optimizes a score function $S(\pi) \in \Re$ such that $S(\pi) > S(\pi_{ref})$. Prior superoptimization settings include inductive program synthesis, where the score function is typically defined as compute or memory cost of the program on a specific hardware or execution environment, conditioned on consistency with respect to a specification (e.g., consistent with unit tests) [28, 26, 25].

Because the space of programs is combinatorially (or infinitely) large, it is important to regularize the optimization problem to some notion of "simple" or "natural programs". This naturally leads to the following maximum a posteriori (MAP) estimation problem [5]:

$$\pi^\star = \arg\max_\pi p_\mathcal{L}(\pi|S) = \arg\max_\pi \underbrace{p_\mathcal{L}(S|\pi)}_{\text{By Execution}} \cdot \underbrace{p_\mathcal{L}(\pi)}_{\text{By Generation}} , \tag{1}$$

where $p_\mathcal{L}(S|\pi)$ expresses the likelihood of a program execution to achieve a certain score, and $p_\mathcal{L}(\pi)$ is a prior distribution on "natural" programs. In practice, we will use a high-quality large language model (LLM) to sample from $p_\mathcal{L}(\pi)$. The term "superoptimization" refers to solving (1) to a superior level than what is practically achievable by human experts.

Superoptimization for Scientific Data Analysis. We model scientific data analysis as a superoptimization problem with three task-specific inputs: (1) a domain-specific dataset of input-output examples $\mathcal{D} := \{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^N$ and a score function $S_\mathcal{D}$ defined on the dataset; (2) an execution environment that includes a production-quality analysis tool such as those depicted in Figure 1; (3) a natural language description provided by domain experts about the data and task, referred to as the Data Prompt and Task Prompt (Figure 2).

All these inputs are either naturally expressible in natural language or can be compiled into executable programs with interpretable natural language meaning. This enables leveraging LLMs to capture the semantic meaning and using them to guide the superoptimization process. LLMs approach the program generation problem as a token prediction problem, directly approximating the program likelihood by training on internet-scale datasets. That is, $p_\mathcal{L}(\pi|S_\mathcal{D}) \approx p_{\text{LLM}}(\langle\pi\rangle|\langle\mathcal{L}\rangle;\langle S_\mathcal{D}\rangle)$, where $\langle\cdot\rangle$ represents a natural language serialization of the respective variables. Because current LLMs cannot reliably predict how their generated code will execute [20], our synthesizer incorporates an agentic workflow [33, 32]: it runs the code, evaluates the output, and feeds the result back to the model. The next sections describe this workflow in detail.

3 Agentic AI framework

To demonstrate the task of superoptimizing scientific analysis workflows, we designed a proof-of-concept agent system, incorporating the design choices commonly seen in the field, including a dual-agent generation-execution feedback loop [7, 34] and a function bank [23, 19]. The key inputs for our framework are: dataset-specific prompts, task-specific prompts, and a library of primitives that act as building blocks for generated functions.

Overall Architecture. The dual-agent feedback loop in Figure 2 combines a code-writing agent that synthesizes new functions based on its prompting and an execution agent to evaluate the functions within a complex workflow and dataset. The functions are composed of OpenCV APIs, adhering to the prompt-defined structure (Section A.11). Execution feedback and evaluation scores are sent back to the code-writing agent to guide revisions to the generated code, forming the feedback loop. The function bank stores generated functions and their associated execution scores. These scores measure how well the functions improve the final output of the scientific workflows, using task-specific metrics (Section 4) formulated into scoring functions. Sampling previous functions and inserting them into the code-writing agent's context can guide later code generations. Although many of these components have been utilized in prior efforts, their specific contributions to achieving superoptimization in scientific analysis workflows remain to be fully elucidated. We investigate this question in the present work.

Prompt Structure. The prompt for the coding agent is structured into four key components: data description, task specification, available primitives, and function samples. The data description details the characteristics of the specific dataset, including attributes such as image type (e.g., RGB or grayscale) and data modality (e.g., dermoscopy, X-ray, or spatial transcriptomics). The task specification outlines the target task, relevant evaluation metrics, and the broader workflow context. Since we are focused on biomedical imaging tasks, the available primitives consist of the OpenCV API functions (Section A.13) that the agent can utilize to generate new functions.

Optimization Procedure. The optimization procedure unfolds in two phases. During Phase 1, the function bank remains unused to encourage the coding agent's exploratory generation of diverse functions. In Phase 2, we strategically sample functions from the bank and incorporate this historical information into the prompt, guiding the coding agent to further refine its function generation based on prior knowledge. The optimization is guided by scores of the final workflow output. The function samples include the highest-performing and lowest-performing functions from the function bank, along with their corresponding performance scores.

Sampling. In each iteration, the coding agent generates multiple functions using the same conversation context. This design choice explicitly motivates the agent to create independent and different functions from the same dependent view of the previous function bank, which implicitly encourages the exploration of a broader functional space. We also run multiple independent rollouts with different random seeds.

Remark on Bottlenecks. Because the tools we optimize for (Section 4) are computationally intensive scientific analysis software, the entire process is typically bottlenecked by tool calls rather than LLM queries. As such, there is less incentive to use faster but less performant LLMs.

4 Case Studies

We consider three biomedical imaging applications as case studies. A production-quality domain-specific tool is provided as part of the workflow execution, together with a labeled dataset and a score function for optimizing the workflow. A baseline is established using a preprocessing function engineered by a domain expert. The three case studies span multiple length scales, from single molecule to cellular to macroscopic.

4.1 Polaris: Single Molecule Spot Detection

This task focuses on detecting sub-pixel fluorescent spots for image-based spatial transcriptomics data (Figure 1, blue box). We use Polaris [13], a comprehensive pipeline for spot detection and gene decoding, focusing specifically on its spot detection component for this case study. The dataset used for this task consists of the held-out validation and test splits from the original training data. All images are from various modalities using different RNA capturing and tagging methods. The expert baseline preprocessing applied involved intensity normalization with clipping (Section A.8). The optimization objective for this task was the maximization of the F1 score.

4.2 Cellpose: Cell Segmentation

The objective of this task is cell instance segmentation on multiple modalities such as whole cell & nucleus (Figure 1, yellow box). The tool we use is the 'cyto-3' model from Cellpose3 [29], a U-Net based network pre-trained for general cell segmentation. The expert baseline applies per-

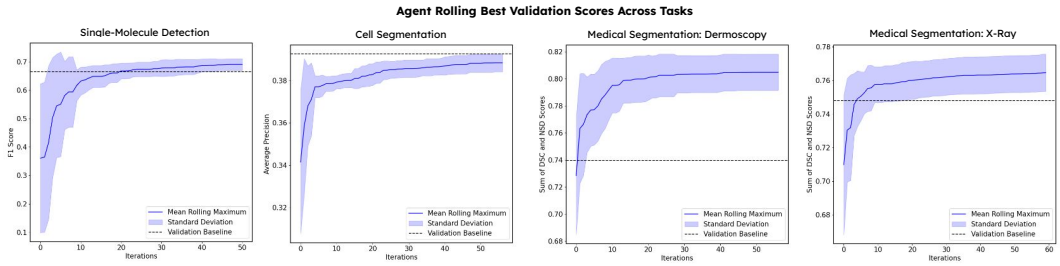


Figure 3: Depicting learning curves on validation set performance. The dotted horizontal line is the validation performance of the expert baseline. We see that average performance continues to improve even when we terminate the experiments, suggesting further improvements are possible.

channel percentile-based min-max normalization (Section A.8). The metric used for optimization and evaluation is the average precision at an Intersection over Union (IoU) threshold of 0.5.

4.3 MedSAM: Medical Segmentation

This task involves medical image segmentation (Figure 1, green box). The tool we use is MedSAM [17], which is an extension of the Segment Anything Model [11] (SAM) specifically adapted for medical imaging domains. We used publicly released subset of validation set from the MedSAM Codabench competition [16], focusing on two specific modalities: Dermoscopy (RGB) and X-ray (grayscale). Each modality has its own expert preprocessing functions: for RGB Dermoscopy images, intensities were scaled using min-max normalization; for grayscale X-ray images (which were then formatted to 3 channels), intensities were first clipped and then rescaled (Section A.8). Evaluation was performed on the full generated test sets for each modality. The optimization and evaluation objective was to maximize the sum of the Normalized Surface Dice (NSD) and Dice Similarity Coefficient (DSC) scores.

5 Experiments

We now present experimental validation on our three case studies. We first demonstrate that agentic superoptimization is indeed achievable using our framework (Section 5.2). We then detail our empirical investigations into the specific design choices influencing the agent's performance, measured by scoring the top performing functions (Section 5.3). We show how inductive biases—originating from the agent's pretraining, prompting strategy, and reliance on function bank history data—can constrain its exploration and performance. We also incorporate results from exhaustive searching using a non-LLM-based conventional optimization approach (AutoML) (Section 5.4) to establish the reliability. Finally, we compare the agent's performance when based on different underlying LLMs, highlighting the influences of LLMs' inherent biases on the search process and final outcomes (Section 5.5).

5.1 Experimental Setup

Agent optimization experiments were conducted on the three case studies described in Section 4: Polaris, Cellpose, and MedSAM. For each task, 20 independent rollouts were performed, with each rollout consisting of 20 iterations. In experiments where the function bank was utilized, it was introduced at the 5th iteration, with top 3 and bottom 3 functions being added to the prompt. We aggregated and selected the final top $K = 1, 10$ functions, evaluated them on the test set, and reported the highest scores. All standard experiments use GPT-4.1 as the base LLM for the code-writing agent.

	Expert	Full System	
		$K = 1$	$K = 10$
Polaris (F1)	0.841	**0.902**	**0.902**
Cellpose (AP@IoU 0.5)	0.403	**0.410**	**0.410**
MedSAM, Dermo (NSD+DSC)	0.836	0.846	**0.881**
MedSAM, X-ray (NSD+DSC)	0.750	0.770	**0.776**

Table 1: Main results on test set. Comparing performance with function selection thresholds $K = 1$ and $K = 10$.

5.2 Agentic Superoptimization vs. Expert Baselines

The main question we ask is: can our agent write code for scientific analysis workflows that outperforms expert-engineered code? This is a non-trivial task given the complexity of the tools used, and the fact that expert baselines are designed over a span of weeks or months.

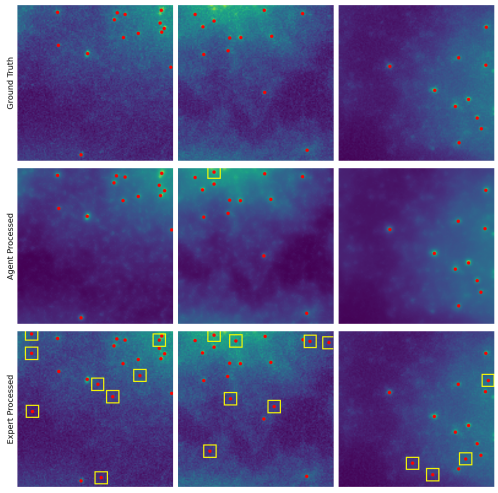
Our results in Table 1 definitively provide a positive answer to this question. Across all case studies, our AI Agent is able to produce workflows that outperform the expert-written baselines on the test set. The minimal improvement of Top-10 performance over Top-1 suggests that the best performing solution on the validation set tends to be the best on the test set as well. Figure 3 shows the learning curve on the validation set, which shows potential for further improvement with more iterations.

Figure 4a shows an example function written by our agent for the Polaris spot detection setting. Compared to the expert-written baseline preprocessing function, which achieves an F1 score of 0.664 on validation and 0.841 on test, the framework-generated preprocessing function achieves an F1 score of 0.741 on validation and 0.902 on test. An improvement on both validation and test indicates that the agent identified more complex image processing methods that better align with the distribution and noise of the dataset than the expert-written baseline. Notably, the agent-generated code in Figure 4a employs an unusual combination of OpenCV's Gaussian blur and Laplacian operator. This unique preprocessing, discovered by the agent, resulted in a reduced false positive rate (Figure 4b).

We re-emphasize that these case studies involve production-quality software workflows. For instance, the function in Figure 4a has been successfully integrated into Polaris production pipeline.

```python
def blurred_laplacian_of_gaussian(images):
    processed_images_list = []
    for img_array in images:
        img = np.copy(img_array)
        img_float32 = cv.normalize(img, None, 0, 1,
cv.NORM_MINMAX).astype(np.float32)
        bilateral = cv.bilateralFilter(img_float32, d=5,
sigmaColor=0.09, sigmaSpace=9)
        gauss = cv.GaussianBlur(bilateral, (3,3), 0)
        lap = cv.Laplacian(gauss, cv.CV_32F, ksize=3)
        abs_lap = np.abs(lap)
        lap_norm = cv.normalize(abs_lap, None, 0, 1,
cv.NORM_MINMAX).astype(np.float32)
        if img_array.ndim == 3 and img_array.shape[2] == 1:
            lap_norm = lap_norm[:, :, np.newaxis]
        processed_images_list.append(lap_norm)
    return np.array(processed_images_list, dtype=np.float32)
```

(a) An agent-generated Polaris preprocessing function, successfully integrated into production.



(b) Detections (red dots) from the agent-proposed function (without library learning) are compared to the Ground Truth and an Expert Baseline. Yellow squares mark incorrect predictions.

Figure 4: An Example of the agent's code generation and visualization on Polaris.

5.3 Impact of design choices

We conduct an array of experiments to understand which agentic design choices result in scientific superoptimization. Due to computational limitations, the MedSAM X-Ray task was excluded from this analysis.

The **Full System** configuration was compared against the following variants:

No Function Bank: Each iteration was provided the same static context, providing no historical information from generated functions and scores.

No Task Prompt: We omitted descriptions about tasks (e.g. "segmentation", "detection"), domain tools, or any associated metric descriptors.

No Data Prompt: We removed any references to the nature of the data such as "medical," "X-Ray," or any domain-specific terminology.

	Full System		−Function Bank		−Task Prompts		−Data Prompts	
	$K=1$	$K=10$	$K=1$	$K=10$	$K=1$	$K=10$	$K=1$	$K=10$
Polaris (F1)	**0.902**	0.902	0.870	**0.926**	0.786	0.925	0.908	0.908
Cellpose (AP@IoU 0.5)	0.410	0.410	**0.412**	**0.412**	0.405	0.405	0.410	0.410
MedSAM, Dermo (NSD+DSC)	0.846	0.881	**0.932**	**0.932**	0.912	0.912	0.918	0.925

Table 2: Test set results on ablations of design choices, comparing $K = 1$ and $K = 10$ thresholds.

Surprisingly, many variants do not degrade performance compared to the default full system setting, as shown in Table 2. Most notably, the removal of the function bank, a common component in agent designs, consistently led to improved performance. Our analysis suggests this happened because the bank restricted the diversity of possible solutions (Section A.2). We also found that removing task-specific or data-specific prompts generally improved predictive performance for MedSAM and Polaris. Conversely, with Cellpose, removing the task-specific prompt decreased performance, while data-prompts had no effect. These results highlight the need for rigorous design and empirical evaluation of AI agent innovations, as they may not actually improve performance, particularly real-world settings such as optimizing production-quality workflows.

5.4 Comparison with AutoML

A common alternative is to use AutoML methods, which effectively amount to some form of random search or Bayesian optimization. Making a completely fair comparison is difficult because the setup in AutoML requires a human expert to specify the allowable function primitive and search space in a more rigid and time-consuming way compared to our agentic prompting. The analysis here is meant to provide a sense of the reliability of each approach.

Setup We compared our agentic framework to standard AutoML using Optuna [2] for hyperparameter optimization. We organized OpenCV functions into families with defined parameter search ranges, allowing 2-4 primitive operations, and optimized using the Tree-structured Parzen Estimator (TPE) sampler. Given the results in Section 5.3, we select the best performing agent variant (from validation set) to compare with AutoML.

Results Following an extensive search of the manually defined optimization space, AutoML's best function sometimes achieves performance on par with our agent's best functions ($K = 1$ and $K = 10$) across tasks, but not reliably so. Notably, the performance of AutoML was particularly poor on tasks exhibiting less saturated performance ceilings, such as Cellpose where AutoML underperformed the expert-written baseline, suggesting greater potential for agentic superoptimization in these scenarios. Moreover, it might be interesting future work to use AutoML as a subroutine in our framework.

	Agent Best		AutoML
	$K=1$	$K=10$	
Polaris (F1)	0.902	**0.926**	0.916
Cellpose (AP@IoU 0.5)	**0.412**	**0.412**	0.339
MedSAM, Dermo (NSD+DSC)	0.932	0.932	**0.936**

Table 3: Comparing Agent system with AutoML on test performance. We selected the best performing variant from Table 2 based on validation performance.

5.5 Impact of LLM choice

Our code-writing LLM replacement ablation demonstrated that the choice of model significantly influences the pre-training inductive bias manifested in code generation. Specifically, the LLaMA-3.3-70B model exhibited poor exploration of the primitive-defined search space and a tendency to generate repetitive functions across tasks, irrespective of context. This limitation resulted in worse performance compared to the agent employing a GPT-4.1 backbone. Moreover, since the overall system is bottlenecked by the tool call (e.g., Polaris, Cellpose, MedSAM), there is little incentive to use faster but less performant LLMs.

6 Related Work

	GPT-4.1		LLaMA 3.3 70B	
	$K = 1$	$K = 10$	$K = 1$	$K = 10$
Polaris	**0.902**	**0.902**	0.881	0.888
Cellpose	**0.410**	**0.410**	0.404	0.404
MedSAM, Dermo	0.846	**0.881**	0.821	0.865

Table 4: Results comparing LLM performance

Program Synthesis & Superoptimization. Superoptimization is the task of finding an optimal program for a given objective, often with the goal of surpassing expert-written baselines. This process is characterized by two main challenges: navigating an enormous search space and satisfying a strict optimality criterion [28, 15]. Superoptimization approaches can be broadly classified into two categories: those that find semantic-preserving modifications and those that discover programs with different semantics [23]. In the first category,[28] introduced performance edits in low-level languages that preserve an algorithm's high-level structure. In the second category, methods like FunSearch [23] and AlphaEvolve [21] have shown that code generated by pretrained LLMs can be evolved to discover more efficient heuristics for problems in mathematics, hardware design, and algorithms. Our work aligns with this second, semantic-altering category, but focuses on the novel application domain of discovering performant algorithms for real-world scientific workflows.

AutoML. Automated Machine Learning (AutoML) automates various pipeline stages, including model hyperparameter optimization [12, 2], feature engineering [10], neural network topology [35, 9], and data cleaning and preprocessing [12, 6, 2]. Despite this breadth, traditional AutoML frameworks operate on a principle of search within a manually defined space. Tools like Optuna [2] require experts to invest significant effort in constructing these search spaces, which acts as a bottleneck by confining the solution to pre-conceived options. Our agentic framework fundamentally diverges from this approach by replacing search with synthesis. Instead of a constrained search space, our system takes simple, high-level instructions and utilizes pretrained LLMs to autonomously synthesize entirely new programs. This eliminates the need for labor-intensive manual configuration and allows for the discovery of solutions beyond the scope of a pre-defined search.

Scientific Analysis Agent. Recent investigations have demonstrated the potential of large language model (LLM) agents in scientific discovery. Existing frameworks aim to automate various facets of the research lifecycle, from the end-to-end process [27] to specialized tasks such as data-driven hypothesis generation [7], rigorous hypothesis validation through falsification [18], the synthesis of interpretable analysis programs [19], and gene perturbation experiment designs [24]. To assess agent capabilities, various benchmarks has been introduced, including MLagentbench [8] for machine learning experimentation and ScienceAgentBench [3] for diverse scientific tasks. Despite these advances in research automation, a significant bottleneck remains: the adaptation of production-level domain tools to achieve expert-level performance, a process that typically requires weeks or months of manual coding. Our work addresses this critical gap by introducing a method to autonomously generate and optimize preprocessing functions that integrate directly into production-level workflows, thereby enabling expert-level performance without extensive manual intervention (Detailed comparison Section A.10)

7 Discussion

Limitation. Our framework's current limitations guide our future work. To improve the inefficient and unreliable hyperparameter generation, we plan to integrate a dedicated lightweight optimizer for the agent to use. Furthermore, while promising, our empirical study was confined to biomedical image preprocessing. We intend to expand our evaluation to a greater variety of tasks, with a specific focus on the pervasive challenge of data standardization. Applying our agent to generate robust transformation pipelines for heterogeneous datasets, like electronic health records or agricultural data, represents a key avenue for increasing its impact across science, health, and society.

Conclusion. We present agentic superoptimization, a method using generative AI to solve a key bottleneck in science: adapting computational tools to new datasets. Our AI agent autonomously wrote and refined pre-processing code for biomedical imaging workflows, outperforming expert-engineered baselines. The success and practical integration of these agent-derived solutions demonstrate a viable path toward accelerating scientific discovery and pushing the boundaries of AI agents in complex, real-world applications.

Acknowledgments

For JJS, this study was supported by the Food and Drug Administration (FDA) of the U.S. Department of Health and Human Services (HHS) as part of a financial assistance award (U01FD008421) totaling $199,907 with 100% percent funded by FDA/HHS. The contents are those of the authors and do not necessarily represent the official views of, nor an endorsement, by FDA/HHS, or the U.S. Government.

YY was supported in part by a gift from Open AI.

References

[1] Pooja Aggarwal, Oishik Chatterjee, Ting Dai, Prateeti Mohapatra, Brent Paulovicks, Brad Blancett, and Arthur De Magalhaes. Codesift: An llm-based reference-less framework for automatic code validation. In *2024 IEEE 17th International Conference on Cloud Computing (CLOUD)*, pages 404–410, 2024.

[2] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '19, page 2623–2631, New York, NY, USA, 2019. Association for Computing Machinery.

[3] Ziru Chen, Shijie Chen, Yuting Ning, Qianheng Zhang, Boshi Wang, Botao Yu, Yifei Li, Zeyi Liao, Chen Wei, Zitong Lu, et al. Scienceagentbench: Toward rigorous assessment of language agents for data-driven scientific discovery. *arXiv preprint arXiv:2410.05080*, 2024.

[4] Noel Crawford, Edward B. Duffy, Iman Evazzade, Torsten Foehr, Gregory Robbins, Debbrata Kumar Saha, Jiya Varma, and Marcin Ziolkowski. Bmw agents – a framework for task automation through multi-agent collaboration, 2024.

[5] Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sable-Meyer, Luc Cary, Lucas Morales, Luke Hewitt, Armando Solar-Lezama, and Joshua B. Tenenbaum. Dreamcoder: Growing generalizable, interpretable knowledge with wake-sleep bayesian program learning, 2020.

[6] Matthias Feurer, Katharina Eggensperger, Stefan Falkner, Marius Lindauer, and Frank Hutter. Auto-sklearn 2.0: Hands-free automl via meta-learning. *arXiv:2007.04074 [cs.LG]*, 2020.

[7] Kexin Huang, Ying Jin, Ryan Li, Michael Y. Li, Emmanuel Candès, and Jure Leskovec. Automated hypothesis validation with agentic sequential falsifications, 2025.

[8] Qian Huang, Jian Vora, Percy Liang, and Jure Leskovec. Mlagentbench: Evaluating language agents on machine learning experimentation. *arXiv preprint arXiv:2310.03302*, 2023.

[9] Haifeng Jin, François Chollet, Qingquan Song, and Xia Hu. Autokeras: An automl library for deep learning. *Journal of Machine Learning Research*, 24(6):1–6, 2023.

[10] James Max Kanter and Kalyan Veeramachaneni. Deep feature synthesis: Towards automating data science endeavors. In *2015 IEEE International Conference on Data Science and Advanced Analytics (DSAA)*, pages 1–10, Campus des Cordeliers, Paris, France, October 2015. IEEE.

[11] Alexander Kirillov, Eric Mintun, Nikhila Ravi, Hanzi Mao, Chloe Rolland, Laura Gustafson, Tete Xiao, Spencer Whitehead, Alexander C. Berg, Wan-Yen Lo, Piotr Dollár, and Ross Girshick. Segment anything. *arXiv:2304.02643*, 2023.

[12] Brent Komer, James Bergstra, and Chris Eliasmith. Hyperopt-sklearn. In Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren, editors, *Automated machine learning: Methods, systems, challenges*, pages 97–111. Springer International Publishing, Cham, 2019.

[13] Emily Laubscher, Xuefei Wang, Nitzan Razin, Tom Dougherty, Rosalind J Xu, Lincoln Ombelets, Edward Pao, William Graf, Jeffrey R Moffitt, Yisong Yue, et al. Accurate single-molecule spot detection for image-based spatial transcriptomics with weakly supervised deep learning. *Cell Systems*, 15(5):475–482, 2024.

[14] Kwanyoung Lee, Hyungjo Byun, and Hyunjung Shim. Cell segmentation in multi-modality high-resolution microscopy images with cellpose. In *Proceedings of The Cell Segmentation Challenge in Multi-modality High-Resolution Microscopy Images*, volume 212 of *Proceedings of Machine Learning Research*, pages 1–11. PMLR, 28 Nov–09 Dec 2023.

[15] Zhengyang Liu, Stefan Mada, and John Regehr. Minotaur: A simd-oriented synthesizing superoptimizer. *Proceedings of the ACM on Programming Languages*, 8(OOPSLA2):1561–1585, 2024.

[16] Jun Ma. Cvpr 2024: Segment anything in medical images on laptop. In *CVPR 2024 Workshop*. OpenReview.net, 2024.

[17] Jun Ma, Yuting He, Feifei Li, Lin Han, Chenyu You, and Bo Wang. Segment anything in medical images. *Nature Commun.*, 15(1):654, January 2024.

[18] Bodhisattwa Prasad Majumder, Harshit Surana, Dhruv Agarwal, Bhavana Dalvi Mishra, Abhijeetsingh Meena, Aryan Prakhar, Tirth Vora, Tushar Khot, Ashish Sabharwal, and Peter Clark. Discoverybench: Towards data-driven discovery with large language models. *arXiv preprint arXiv:2407.01725*, 2024.

[19] Utkarsh Mall, Cheng Perng Phoo, Mia Chiquier, Bharath Hariharan, Kavita Bala, and Carl Vondrick. Disciple: Learning interpretable programs for scientific visual discovery, 2025.

[20] Ansong Ni, Miltiadis Allamanis, Arman Cohan, Yinlin Deng, Kensen Shi, Charles Sutton, and Pengcheng Yin. Next: Teaching large language models to reason about code execution, 2024.

[21] Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco JR Ruiz, Abbas Mehrabian, et al. Alphaevolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025.

[22] Zeeshan Rasheed, Muhammad Waseem, Aakash Ahmad, Kai-Kristian Kemell, Wang Xiaofeng, Anh Nguyen Duc, and Pekka Abrahamsson. Can large language models serve as data analysts? a multi-agent assisted approach for qualitative data analysis, 2024.

[23] Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M Pawan Kumar, Emilien Dupont, Francisco JR Ruiz, Jordan S Ellenberg, Pengming Wang, Omar Fawzi, et al. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475, 2024.

[24] Yusuf Roohani, Andrew Lee, Qian Huang, Jian Vora, Zachary Steinhart, Kexin Huang, Alexander Marson, Percy Liang, and Jure Leskovec. Biodiscoveryagent: An ai agent for designing genetic perturbation experiments, 2025.

[25] Raimondas Sasnauskas, Yang Chen, Peter Collingbourne, Jeroen Ketema, Jubi Taneja, and John Regehr. Souper: A synthesizing superoptimizer. 2017.

[26] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. In *ACM SIGARCH computer architecture news*, volume 41, pages 305–316. ACM, 2013. Number: 1.

[27] Samuel Schmidgall, Yusheng Su, Ze Wang, Ximeng Sun, Jialian Wu, Xiaodong Yu, Jiang Liu, Michael Moor, Zicheng Liu, and Emad Barsoum. Agent laboratory: Using llm agents as research assistants. *arXiv preprint arXiv:2501.04227*, 2025.

[28] Alexander Shypula, Aman Madaan, Yimeng Zeng, Uri Alon, Jacob Gardner, Milad Hashemi, Graham Neubig, Parthasarathy Ranganathan, Osbert Bastani, and Amir Yazdanbakhsh. Learning performance-improving code edits, 2024.

[29] Carsen Stringer and Marius Pachitariu. Cellpose3: one-click image restoration for improved cellular segmentation. *Nature Methods*, pages 1–8, 2025.

[30] Ana Trisovic, Matthew K. Lau, Thomas Pasquier, and Mercè Crosas. A large-scale study on research code quality and execution. *Scientific Data*, 9(1):60, February 2022. Publisher: Nature Publishing Group.

[31] Hanchen Wang, Tianfan Fu, Yuanqi Du, Wenhao Gao, Kexin Huang, Ziming Liu, Payal Chandak, Shengchao Liu, Peter Van Katwyk, Andreea Deac, et al. Scientific discovery in the age of artificial intelligence. *Nature*, 620(7972):47–60, 2023.

[32] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering, 2024.

[33] Shunyu Yao. *Language Agents: From Next-Token Prediction to Digital Automation*. PhD thesis, Princeton University, 2024.

[34] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023.

[35] Lucas Zimmer, Marius Lindauer, and Frank Hutter. Auto-pytorch tabular: Multi-fidelity metalearning for efficient and robust autodl. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 3079 – 3090, 2021. also available under https://arxiv.org/abs/2006.13799.

A Appendix

The sections of our appendix are organized as follows:

- Section A.1: Qualitative results where we visualized the expert and agent processed images with downstream task prediction overlaid.
- Section A.2: Diversity analysis of primitives between full-system and no-function-bank settings.
- Section A.3: Polaris author's comment on the agent-generated function.
- Section A.4: Experiments where we run 2x iterations compared to the standard setup.
- Section A.5: Experiments with a different split of data.
- Section A.6: Experiments with a ReAct-style variant.
- Section A.7: Statistics of the primitives used by top and bottom functions by each task.
- Section A.8: Expert baseline functions and their git hisotry analysis.
- Section A.9: Top-1 Agent generated functions.
- Section A.10: Comparison of our method against related work.
- Section A.11: Prompts and data split details, including which prompts we ablated during design choice experiments.
- Section A.12: Computational requirements for each task.
- Section A.13: A list of primitives used in the preprocessing functions.

A.1 Visualizations of Top Performing Preprocessing Functions

A.1.1 Polaris

See Figure 4b.

A.1.2 Cellpose

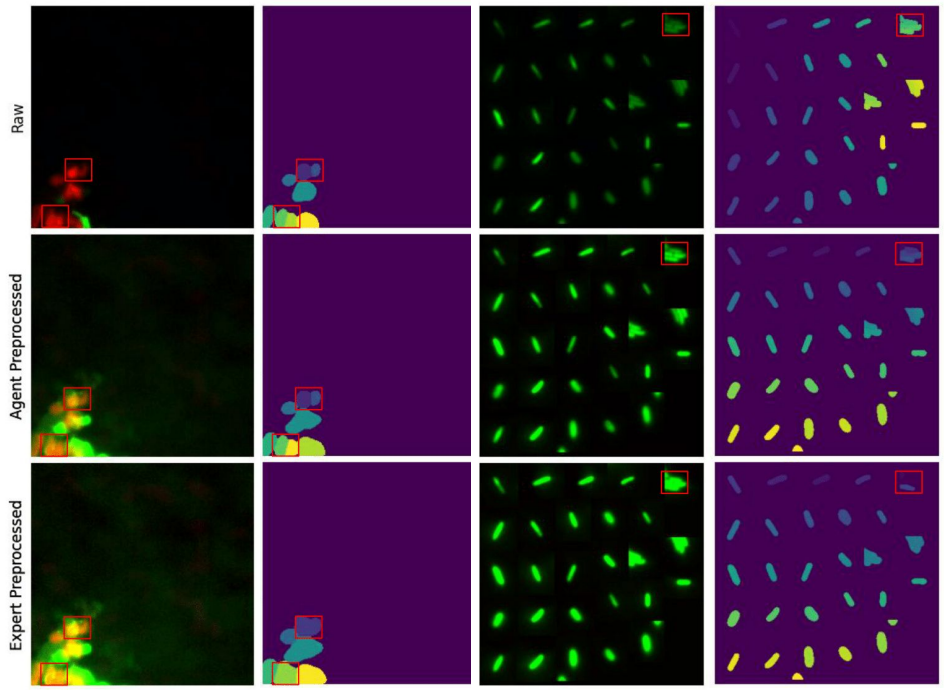


Figure 5: Segmentation masks from the Top-1 Function returned on Cellpose (without library learning), compared to the Raw Image and the Expert Baseline on the two example images. Highlighted in red boxes are places within the image where the agent's returned preprocessing function improves performance by either preventing merges of multiple structures into single structures or enhanced segmentation mask accuracy.

A.1.3 MedSAM

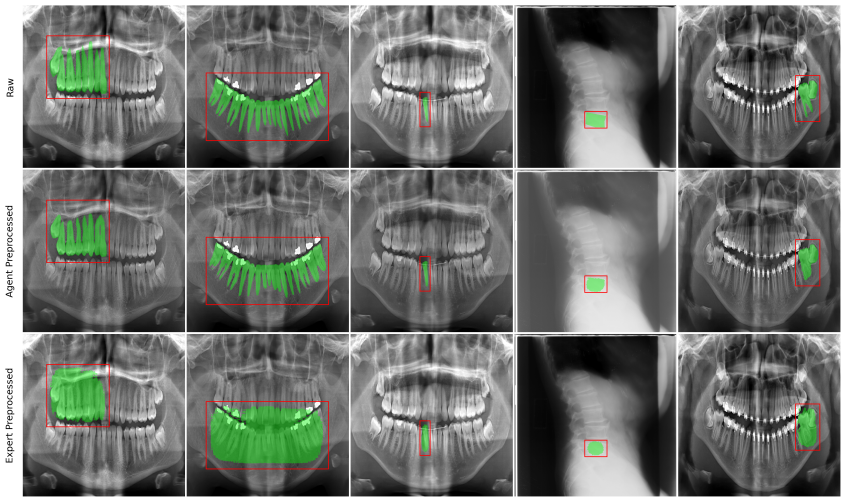


Figure 6: MedSAM X-ray images with mask overlays. The Raw row shows unprocessed images with their corresponding ground truth masks. The Agent Preprocessed row displays images preprocessed using the Top-1 function returned on MedSAM X-ray images (without library learning), overlaid with the predicted masks. The Expert Preprocessed row presents images processed using the methods described in the original MedSAM paper and official repository, also overlaid with their predicted masks.

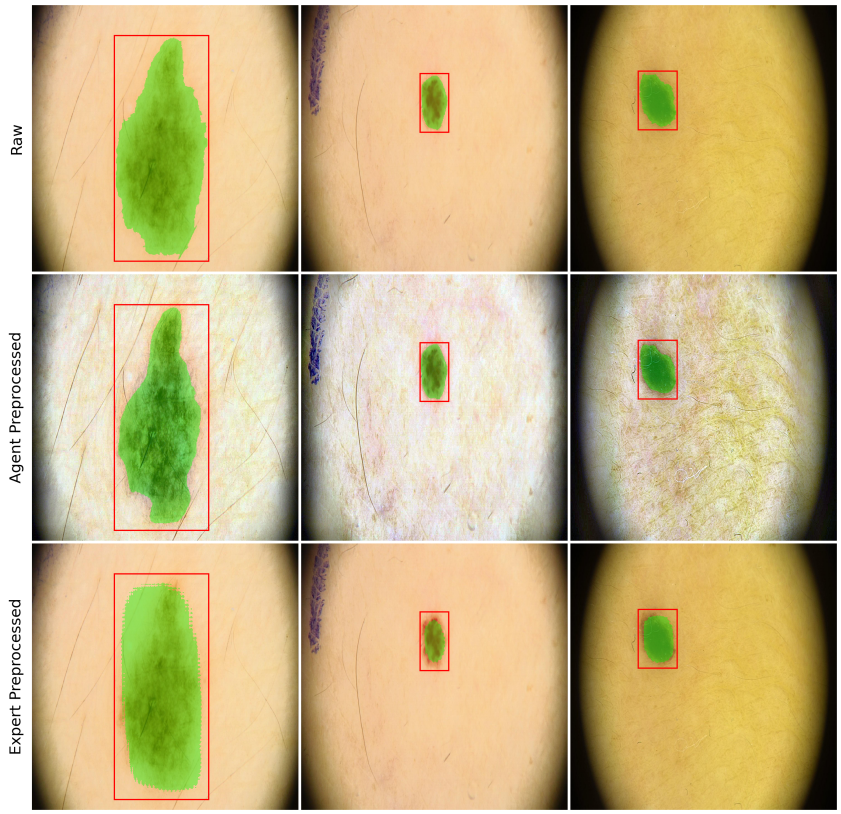


Figure 7: MedSAM Dermoscopy images with mask overlays. The Raw row shows unprocessed images with their corresponding ground truth masks. The Agent Preprocessed row displays images preprocessed using the Top-1 function returned on MedSAM Dermoscopy images (without library learning), overlaid with the predicted masks. The Expert Preprocessed row presents images processed using the methods described in the original MedSAM paper and official repository, also overlaid with their predicted masks.

A.2 Diveristy Analysis of primitives with full-system and no-function-bank settings

We conducted diversity analysis of primitives on Polaris: with function banks, common methods like 'normalize' and 'GaussianBlur' were overrepresented (about 30% and 17.5% respectively), while without them, the top two functions were less skewed (about 22% and 12%), allowing for greater exploration and the proposal of more effective, diverse functions like 'CLAHE' or 'Laplacian' filters. Further, either of the two functions can be found in all 20 of the top 1 functions across 20 rollouts with function banks, whereas without function banks they represent only 14 of the top 1 functions. This demonstrates that function banks, counter-intuitively, bias exploration and lead to sub-optimal results in our setting.

A.3 Polaris Author's Comment on Agent-generated Function

> *This is a great improvement! It makes sense that the bilateral filtering is removing noise in an edge-aware way, which would preserve the structure of the spot intensities. Followed by Gaussian blurring, which would more uniformly remove high frequency noise. It's likely helping to have both because the objects we're detecting are so small and easily altered when the parameters of a particular filtering method are too aggressive. The final LoG step makes sense too as a final edge detection step. Looks great and so glad it's improved the spot detection metrics!*

A.4 Extended Iteration Experiments

We doubled the length of each rollout for experiments in the standard setting to observe the effect on final top-1 and top-10 validation and test performance (Table. 5). As expected, the learning curve keeps improving. And the top-1 and top-10 test scores are the same for both tasks, indicating that improvement is better aligned between test and validation sets when we allow a larger compute budget.

	Val		**Test**	
	$K = 1$	$K = 10$	$K = 1$	$K = 10$
Cellpose	0.3969	0.3969	0.4109	0.4109
Polaris	0.9143	0.9143	0.7339	0.7339

Table 5: Top-1 and Top-10 validation and test scores with extended iteration experiments.

A.4.1 Polaris

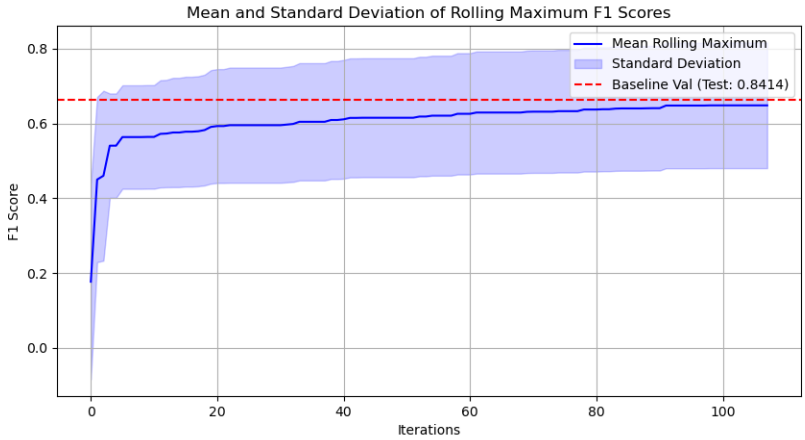


Figure 8: Learning curve for extended rollouts for Polaris

A.4.2 Cellpose

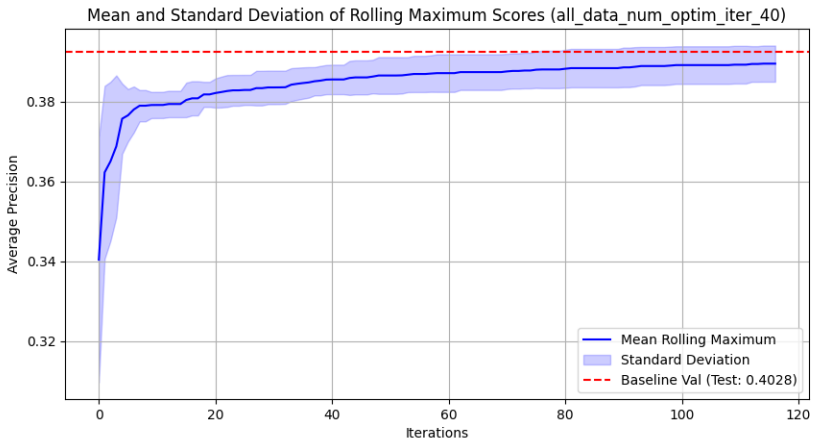


Figure 9: Learning curve for extended rollouts for Cellpose

A.5 Experiments with different splits of data

To confirm that the agent framework is robust and not affected by different data splits, we reran the Cellpose experiment with a new data split. The results showed consistent scores across all splits, demonstrating the framework's reliability.

	Expert	**Test**	
		$K = 1$	$K = 10$
New Split	0.401	0.411	0.411
Old Split	0.403	0.410	0.410

Table 6: Cellpose test scores on different data splits with the full-system setting.

A.6 Experiments investigating the role of reasoning in superoptimization

To determine if enhanced reasoning capabilities improve performance in scientific superoptimization, we created a ReAct-style version of our system. This variant included a reasoning trace and sampled only the most recent function history. This approach allows our framework to rigorously test why certain agent strategies succeed or fail in this challenging domain. Our results demonstrated that incorporating a ReAct-style reasoning mechanism did not yield additional benefits or performance gains for this task. Similarly, we ran an experiment using OpenAI's "o3" reasoning model and found that it did not offer a significant advantage for this superoptimization task either.

	Expert	**Test**	
		$K = 1$	$K = 10$
ReAct-style (gpt-4.1)	0.841	0.886	0.904
Full-system (o3)	0.841	0.839	0.905
Full-system (gpt-4.1)	0.841	0.902	0.902

Table 7: Performance of reasoning-enhanced variants compared to the full system on the Polaris task.

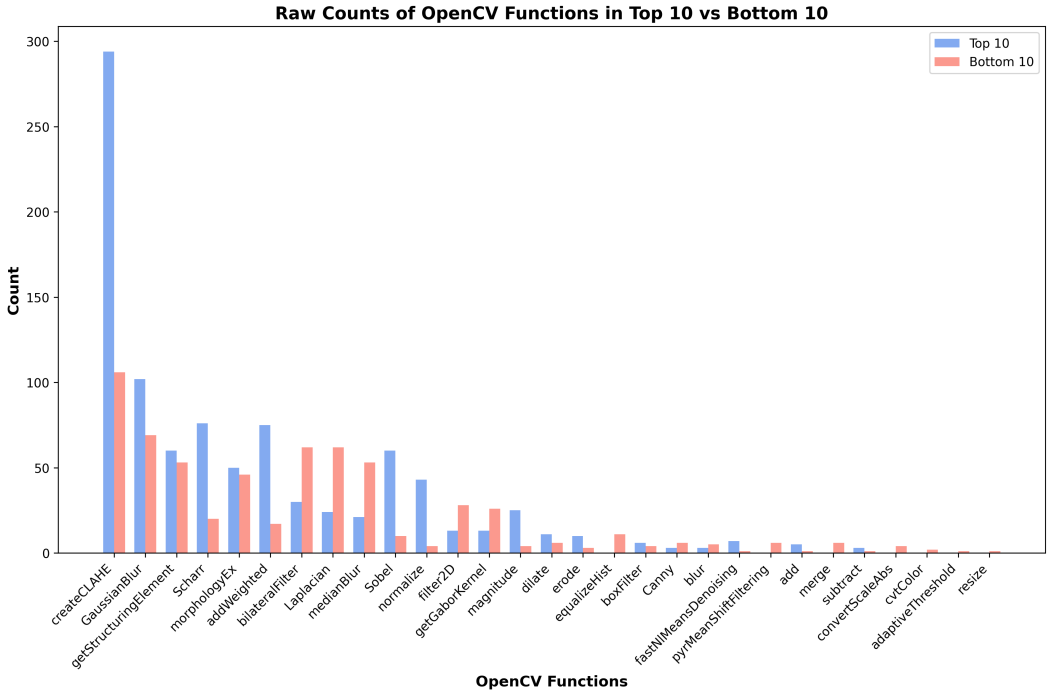


Figure 10: Cell Segmentation: Distribution of function primitives used in the aggregated top-10 or bottom-10 functions returned by agentic optimization (across 20 rollouts).

A.7 Statistics of primitives used in Top-K and Worst-K Functions

By analyzing the function abstract syntax trees, we show the distributions of functional primitives used in the best and worst performing preprocessing functions found by our agentic optimization pipeline. Figure 10 shows the distributions found for Cell Segmentation (Cellpose) by aggregating the top 10 or bottom 10 performing (by validation performance) functions returned in each rollout (20 rollouts).

Compare the distribution of best/worst functions in the Cellpose task with the distribution of best/worst functions in Fig 11 for the Polaris task and Fig 12 for the MedSAM task, collected using the same method above.

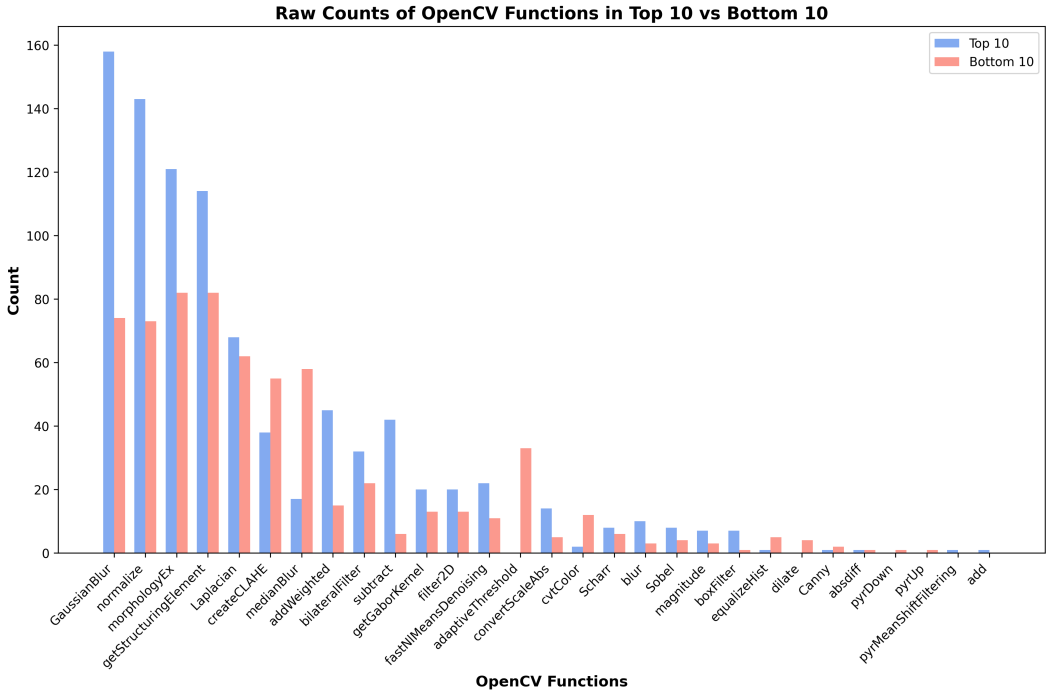


Figure 11: Polaris: Distribution of function primitives used in the aggregated top-10 or bottom-10 functions returned by agentic optimization (across 20 rollouts).

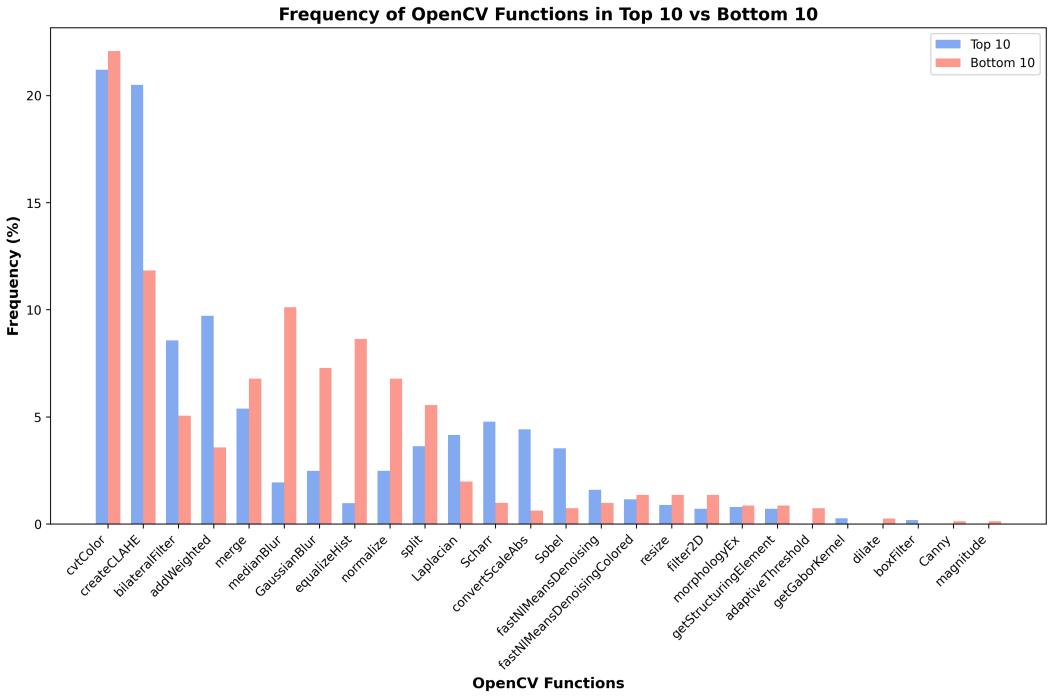


Figure 12: Medical Segmentation: Distribution of function primitives used in the aggregated top-10 or bottom-10 functions returned by agentic optimization (across 20 rollouts).

A.8 Expert functions

A.8.1 Polaris expert baseline function

The function below is the default expert baseline function from the Polaris GitHub repository, as detailed in the Polaris paper [13]. Given that the training data underwent min-max normalization — a process also applied by the agent-generated function — the effective baseline for our experiments is the identity function.

Expert Polaris Preprocessing Function

```
import numpy as np
def min_max_normalize(image, clip=False):
    if not np.issubdtype(image.dtype, np.floating):
        logging.info('Converting image dtype to float')
    image = image.astype('float32')

    if not len(np.shape(image)) == 4:
        raise ValueError('Image must be 4D, input image shape was'
                         ' .'.format(np.shape(image)))

    for batch in range(image.shape[0]):
        for channel in range(image.shape[-1]):
            img = image[batch, ..., channel]

            if clip:
                img = np.clip(img, a_min=np.percentile(img, 0.01),
                a_max=np.percentile(img, 99.9))

            min_val = np.min(img)
            max_val = np.max(img)
            normal_image = (img - min_val) / (max_val - min_val)

            image[batch, ..., channel] = normal_image
    return image
```

A.8.2 Cellpose expert baseline function

Adapted and simplified from the Cellpose GitHub repository, as described in the Cellpose3 paper [29].

Expert Cellpose Preprocessing Function

```
import numpy as np
def normalize99(Y, lower=1, upper=99, copy=True, downsample=False):
    '''Normalize the image so that 0.0 corresponds to the 1st percentile
           and 1.0 corresponds to the 99th percentile.

       Args:
           Y (ndarray): The input image (for downsample, use [Ly x Lx]
               or [Lz x Ly x Lx]).
           lower (int, optional): The lower percentile. Defaults to 1.
           upper (int, optional): The upper percentile. Defaults to 99.
           copy (bool, optional): Whether to create a copy of the input image.
               Defaults to True.
           downsample (bool, optional): Whether to downsample image to compute
               percentiles. Defaults to False.
       Returns:
           ndarray: The normalized image.'''
    # Create a copy of the input if required
    X = Y.copy() if copy else Y
    X = X.astype("float32") if X.dtype!="float64" and X.dtype!="float32" else X
```

```
    # Downsample to calculate percentiles more efficiently for large images
    if downsample and X.size > 224**3:
        nskip = [max(1, X.shape[i] // 224) for i in range(X.ndim)]
        nskip[0] = max(1, X.shape[0] // 50) if X.ndim == 3 else nskip[0]
        slc = tuple([slice(0, X.shape[i], nskip[i]) for i in range(X.ndim)])
        x01 = np.percentile(X[slc], lower)
        x99 = np.percentile(X[slc], upper)
    else:
        x01 = np.percentile(X, lower)
        x99 = np.percentile(X, upper)

    # Normalize the image between the percentiles
    if x99 - x01 > 1e-3:
        X -= x01
        X /= (x99 - x01)
    else:
        X[:] = 0

    return X
```

A.8.3 MedSAM expert baseline function

Adapted and simplified from the MedSAM GitHub repository, as described in the MedSAM paper [17].

Expert MedSAM Preprocessing Function

```
import numpy as np
def preprocess_images(images, is_rgb):
    resized_imgs = images.raw
    for i in range(len(resized_imgs)):
        img_np = resized_imgs[i]
        if is_rgb:
            resized_imgs[i] = np.uint8((img_np - img_np.min()) /
            (np.max(img_np) - np.min(img_np)) * 255.0)

        else:
            lower_bound, upper_bound = np.percentile(img_np[img_np > 0],
            0.5), np.percentile(img_np[img_np > 0], 99.5)

            img_np_pre = np.clip(img_np, lower_bound, upper_bound)

            img_np_pre = (img_np_pre - np.min(img_np_pre)) / (np.max(img_np_pre)
            - np.min(img_np_pre)) * 255.0

            img_np_pre[img_np == 0] = 0

            resized_imgs[i] = np.uint8(img_np_pre)
    return ImageData(raw=resized_imgs, batch_size=images.batch_size)
```

A.8.4 Git history analysis of expert functions

We analyzed the git R&D history of these packages to quantify the efforts experts put into building these baseline functions:

Polaris: Its Git history (May 29, 2021 to September 15, 2021, across 12 commits) shows an evolution over 6 months including changes like adding clipping and reordering functions, resulting in a refined version with 2 preprocessing options and 2 hyperparameters.

MedSAM: While direct R&D commit history was not available, MedSAM incorporates custom preprocessing functions for various modalities (CT, MR, grey, RGB). The substantial codebase

dedicated to these functions (255 lines of code) indicates significant expert investment in tailoring these for specific medical imaging challenges.

Cellpose: The human optimization process spans over three years (February 1, 2020 to March 8, 2023) along with model development. Changes include normalization for 3D data, adjustments to tiling strategies, and the addition or removal of sharpen/smooth operations. The final version provides 9 hyperparameters.

A.9 Agent Top-1 functions

A.9.1 Polaris

Top-1 Agent Generated Function (Polaris)

```
def preprocess_images(images):
    """
    Preprocessing 2: Bilateral Filter + Laplacian Edge Enhancement.
    Output is (H, W, 1) float32 in [0,1].
    """
    processed_images_list = []
    for img_array in images.raw:
        img_array = np.copy(img_array)
        squeeze = False
        if img_array.ndim == 3 and img_array.shape[-1] == 1:
            img = img_array[:, :, 0]
            squeeze = True
        else:
            img = img_array

        img8 = (img / img.max() * 255).astype(np.uint8)
            if img.max() > 1 else (img * 255).astype(np.uint8)
        img_bilat = cv.bilateralFilter(img8, d=5, sigmaColor=75, sigmaSpace=75)
        img_lap = cv.Laplacian(img_bilat, cv.CV_16S, ksize=3)
        img_lap = cv.convertScaleAbs(img_lap)
        img_enhanced = cv.addWeighted(img_bilat, 0.8, img_lap, 0.2, 0)
        img_enhanced = img_enhanced.astype(np.float32) / 255.0
        if squeeze:
            img_enhanced = img_enhanced[..., np.newaxis]
        processed_images_list.append(img_enhanced)
    output_data = ImageData(raw=processed_images_list,
        batch_size=images.batch_size)
    return output_data
```

A.9.2 Cellpose

Top-1 Agent Generated Function (Cell Segmentation)

```
def preprocess_images(images):
        # Function 1: Apply CLAHE to channel 0 (nucleus), controlled contrast
            stretching on channel 1 (cytoplasm), channel 2 unchanged
        processed_images_list = []
        for img_array in images.raw:
            img = np.copy(img_array)
            output = np.zeros_like(img)
            # Channel 0: CLAHE (for better nucleus contrast)
            ch0 = (img[..., 0] * 255).astype(np.uint8)
            clahe = cv.createCLAHE(clipLimit=2.0, tileGridSize=(8, 8))
            ch0_eq = clahe.apply(ch0)
            output[..., 0] = ch0_eq.astype(np.float32) / 255.0
            # Channel 1: Contrast stretching (clip 1st & 99th percentiles)
            ch1 = img[..., 1]
```

```
            p1, p99 = np.percentile(ch1, 1), np.percentile(ch1, 99)
            ch1_stretch = np.clip((ch1 - p1) / (p99 - p1 + 1e-6), 0, 1)
            output[..., 1] = ch1_stretch
            # Channel 2: unchanged
            output[..., 2] = img[..., 2]
            processed_images_list.append(output)
        return ImageData(raw=processed_images_list)
```

A.9.3 MedSAM

Top-1 Agent Generated Function (Medical Segmentation)

```
def preprocess_images(images: ImageData) -> ImageData:
    # Variant: Sharpen after CLAHE + GaussianBlur, then normalize
    processed_images_list = []
    for img in images.raw:
        img = np.copy(img)
        img = cv.resize(img, (1024, 1024), interpolation=cv.INTER_AREA)
        img = cv.GaussianBlur(img, (3, 3), 0)
        if img.shape[2] == 3:
            clahe = cv.createCLAHE(clipLimit=2.0, tileGridSize=(8,8))
            channels = cv.split(img)
            eq_channels = [
                clahe.apply(cv.normalize(c, None, 0, 255,
                cv.NORM_MINMAX).astype(np.uint8))
                for c in channels
            ]
            img = cv.merge(eq_channels)
        else:
            img = cv.equalizeHist(cv.normalize(img, None, 0, 255,
            cv.NORM_MINMAX).astype(np.uint8))

            img = np.expand_dims(img, axis=-1)
        # Sharpening kernel
        kernel = np.array([[0, -1, 0],
                           [-1, 5, -1],
                           [0, -1, 0]])
        img = cv.filter2D(img, -1, kernel)
        img = img.astype(np.float32) / 255.0
        processed_images_list.append(img)

    output_data = ImageData(raw=processed_images_list,
    batch_size=images.batch_size)
    return output_data
```

A.10 Comparison of our method against related work

Here is the detailed comparison between our work (agentic superoptimization) and conventional ML agent tasks (like MLAgentBench [8]):

- **Primary Goal**
  - **MLAgentBench**: Optimize the entire ML experimentation process in a standard, fixed setting for quickly testing and iterating on ML models, not translatable to real-world scientific deployment.
  - **Our Work**: Superoptimize a single component within an existing, fixed, production-level scientific workflow that are used by scientists for real-world scientific discovery (already optimized by experts with weeks/months of development time, thus superoptimization).
- **Nature of the Task**
  - **MLAgentBench**: Whitebox model development with full flexibility - the agent builds or modifies the core predictive model to improve performance on a dataset. The agent generally has full control and "white-box" access to the code it writes. It defines the entire workflow, from

Table 8: Comparison of our framework against related work in automated scientific discovery. Our work is the first to combine code optimization with the use of production-level tools to achieve expert performance in deployed scientific workflows.

	Code Optimization	Uses Production-level Tools	Solves Real-World Tasks	Reaches Expert Performance	Integrates into Deployed Workflows
ScienceAgentBench [3]	✗	✓	✓	✗	✗
DiscoveryBench [18]	✗	✗	✓	N/A	✗
MLagentbench [8]	✓	✗	✓	✗	✗
Agent Laboratory [27]	✓	✗	✗	✓	✗
DiSciPLE [19]	✓	✗	✓	✓	✗
Popper [7]	✗	✗	✓	✓	✗
Ours	✓	✓	✓	✓	✓

the model architecture to the training loop. This setup is designed so that most actions lead to an improvement over a naive baseline.

- **Our work**: Blackbox optimization with a challenging optimization terrain - the agent composes a new preprocessing function from a library of primitives (like OpenCV) to improve the performance of a downstream scientific tool (like Polaris, Cellpose, MedSAM) that it cannot change. It tries to optimize its component for a downstream tool that acts as a black box. The agent cannot see inside or modify this tool. Because the workflow has already been heavily optimized by experts, most changes the agent makes will result in a decrease in performance.

- **The Definition of Success**
  - **MLAgentBench**: Predictive Performance - success is defined by outperforming a simple baseline given by the starter code, as mentioned above.
  - **Our work**: Production Integration - the solution must meet the real-world expert-performance level, which we validate using metrics, by integrating the tool into a real-world production pipeline.

A.11 Prompts

A.11.1 LLM Agent domain-generic task prompt

Experiment specific or run specific experimental settings (seed, GPU ID) are marked in <red>. Per-experiment prompts are included below.

Agent Header

Agent Pipeline Seed <3280387012>

Preprocessing Function API

```
# Necessary imports for any function's logic (if any)
# Do not import ImageData in the functions, it is already imported in the
# environment
# All preprocessing function names should be of the form
# preprocess_images_i
import cv2 as cv
def preprocess_images_i(images: ImageData) -> ImageData:
    # Function logic here
    processed_images_list = []
    # This iterates through each image to avoid modifying the
    # the original image
    for img_array in images.raw:
        img_array = np.copy(img_array)  # Create a copy to avoid
        processed_img = img_array  # Replace with actual processing
        processed_images_list.append(processed_img)
    output_data = ImageData(raw=processed_images_list,
```

```
                                batch_size=images.batch_size)
    return output_data
```

About the Dataset

<Experiment specific Data Details here. Refer below for prompts used.>

Task Details

All of you should work together to write three preprocessing functions that improve segmentation performance using OpenCV functions (APIs provided). It might make sense to start the process with small preprocessing functions, and then build up to more complex functions depending on the performance of the previous functions.

1. Based on previous preprocessing functions and their performance (provided below), suggest three new unique preprocessing functions using OpenCV functions (APIs provided below). Successful strategies can include improving upon high performing functions (including tuning the parameters of the function), or exploring the image processing space for novel or different image processing approaches. You can feel free to combine OpenCV functions or suggest novel combinations that can lead to improvements, or modify the parameters of the existing extremely successful functions.
2. Remember, the images after preprocessing must still conform to the format specified in the ImageData API. Maintenance of channel identity is critical and channels should not be merged.
3. The environment will handle all data loading, evaluation, and logging of the results. Your only job is to write the preprocessing functions.
4. For this task, if all three functions are evaluated correctly, only one iteration is allowed, even if the performance is not satisfactory.
5. Do not terminate the conversation until the new preprocessing functions are evaluated and the numerical performance metrics are logged.
6. Extremely important: Do not terminate the conversation until each of the three new preprocessing functions are evaluated AND their results are written to the function bank.
7. Recall, this is a STATELESS kernel, so all functions, imports, etc. must be provided in the script to be executed. Any history between previous iterations exists solely as provided preprocessing functions and their performance metrics.
8. Do not write any code outside of the preprocessing functions.
9. Do not modify the masks under any circumstances.
10. The preprocessing functions written must return an ImageData object with each image in the batch having the same image resolution (H,W) as the original image.

Task Metrics Details

<Experiment specific Task Metrics Details here.>

Function Bank Sample

Function bank history will be shown after iteration 5, you are currently on iteration 0 of 20

OpenCV Function API

```
cv.bilateralFilter(src, d, sigmaColor, sigmaSpace[, dst[, borderType]]) ->dst
Applies the bilateral filter to an image.

cv.GaussianBlur(src, ksize, sigmaX[, dst[, sigmaY[, borderType]]]) ->dst
Blurs an image using a Gaussian filter.

                                        ⋮
cv.merge(mv[, dst]) -> dst
Creates one multi-channel array out of several single-channel ones.
```

The ImageData Class

```
Framework-agnostic container for batched image data. Handles variable image
resolutions

This class provides a standardized structure for storing and managing batched
image data along with related annotations and predictions.
Data is internally converted to lists of arrays for flexibility with varying
image sizes.

Attributes:
    raw (Union[List[np.ndarray], np.ndarray]): Raw image data, can be provided
        as either a list of arrays or a numpy array. Each image should have
        shape (H, W, C).

    batch_size (Optional[int]): Number of images to include in the batch. Can
        be smaller than the total dataset size. If None, will use the full
        dataset size.

    image_ids (Union[List[int], List[str], None]): Unique identifier(s) for
    images in the batch as a list. If None, auto-generated integer IDs
    [0,1,2,...] will be created.

    channel_names (Optional[List[str]]): Names of imaging channels in order
    matching raw data channels. Length must equal number of channels.

    masks (Optional[Union[List[np.ndarray], np.ndarray]]): Ground truth
    segmentation masks. Integer-valued arrays where 0 is background and positive
    integers are unique object identifiers. Each mask should have shape
    (H, W, 1) or (H, W).
```

Important: When returning processed images, they must maintain the same dimensions (H,W) as the original images and preserve channel order.

Additional Notes

- Always check the documentation for the available APIs before reinventing the wheel
- Use GPU <0> for running the pipeline, set `cuda: 0` in the code snippet!
- You only have 20 rounds of each conversation to optimize the preprocessing function.
- Don't suggest trying larger models as the model size is fixed.
- THE PROVIDED EVALUATION PIPELINE WORKS OUT OF THE BOX, IF THERE IS AN ERROR IT IS WITH THE PREPROCESSING FUNCTION

Code Writer Agent Instructions

You are an experienced Python developer specializing in scientific data analysis. Your role is to write, test, and iterate on Python code to solve data analysis tasks. The environment is installed with the necessary libraries.
You write code using Python in a STATELESS execution environment, so all code must be contained in the same block. In the environment, you can:
Notes:

- Write code in Python markdown code blocks:
  “‘python
  def preprocess_images_i(image_data: ImageData) -> ImageData: Code example. All code must be written within this function.
  return image_data
  ”’
- CRITICAL: You must define three functions at once, and they must be named ‘preprocess_images_i‘ where ‘i‘ starts at 1 and ranges to 3. The functions must follow the provided Preprocessing Functions API. All operations must be performed within the functions, and no inner functions should be defined (construct all operations within the functions).
- Code outputs will be returned to you
- Feel free to document your thought process and exploration steps.
- Remember that all images processed by your written preprocessing functions will directly be converted into ImageData objects. So, double-check that the preprocessed image dimensions align with the dimension requirements listed in the ImageData API documentation
- Make sure each response has exactly one code block containing all the code for the preprocessing functions, and that the code block ONLY contains the code for the preprocessing functions. Do not include any mock code for data loading or evaluation.
- All three functions must be defined at once, and they must be named ‘preprocess_images_i‘ where ‘i‘ starts at 1 and ranges to 3. The functions must follow the provided Preprocessing Functions API.
- Once metrics have been evaluated for all three preprocessing functions successfully, please print them out for each function in the format: preprocess_images_<i>: <metric>: <score>. You may only emit "TERMINATE" once all three preprocessing functions have been evaluated and their metrics printed successfully.
- If metrics are not correctly returned for any of the three preprocessing functions and you need to fix the underlying errors, output all three revised functions in a single markdown block. On the other hand, if all functions were successfully evaluated, do not continue iterating, and emit "TERMINATE".
- For generating numbers or variables, you will need to print those out so that you can obtain the results
- Write "TERMINATE" when the task is complete

A.11.2 Polaris

Data prompt We provide a data-specific prompt detailing the cell stain spots modality and the single-channel dimensionality of the dataset. Specific instructions regarding index order of the dimensions (length, width, channel, batch) are given in the prompt.

This is a single-channel cell spot detection dataset. IMPORTANT: The cell images have dimensions (B, L, W, C) = (batch, length, width, channel).

Blue boxes mark text deleted for the data prompt ablation. Peach underlined text marks text deleted for the task prompt ablation.

About the Dataset

This is a single-channel cell spot detection dataset. IMPORTANT: The cell images have dimensions (B, L, W, C) = (batch, length, width, channel) .

Task prompt The LLM is prompted to generate new image preprocessing functions using OpenCV functions, and encouraged to consider previously generated functions when writing a new one. We provide a short description of the evaluation metrics, principally classification loss created from one-hot encoded detections, regression loss created from the distance of predictions to ground truth spots, and the F1 score of predicted spots. The LLM is prompted to optimize on the F1 score.

Polaris Task Metrics Details

Task Metrics Details:

class_loss: loss from one-hot encoded 2D matrix, where 1 is a spot and 0 is not a spot

regress_loss: loss 2D matrix where each entry is distance from a predicted spot

f1_score: Mean F1 score of predicted spots

Data Split For the optimization procedure, we used 95 images from the validation set. The performance of functions was then evaluated on the test set, comprising 94 images. Both validation and test images have a fixed size of 128 by 128 pixels. Ground truth for Polaris consists of a list of point coordinates.

A.11.3 Cellpose

Data prompt The agent is provided with a prompt describing that the dataset is a heterogenous biological image dataset with a focus on biological microscopy images, including cells. It also includes information that the channels are ordered as: nuclear, cytoplasmic, and empty. The LLM is also provided with details about how to conform to the provided image data API (resolutions must not change and channels must not be reordered, or compressed to grayscale).

Blue boxes mark text deleted for the data prompt ablation. Peach underlined text marks text deleted for the task prompt ablation.

About the Dataset

This is a three-channel image dataset for biological segmentation, consisting of images from different experiments and different settings - a heterogenous dataset of many different object types. There is a particular focus on biological microscopy images, including cells, sometimes with nuclei labeled in a separate channel.

The images have pixel values between 0 and 1 and are in float32 format.

Channel[0] is the nucleus, channel[1] is the cytoplasm, and channel[2] is empty, however not all images have any nuclear data.

Our goal is to improve the segmentation performance of the neural network by using OpenCV preprocessing functions to improve the quality of the images for downstream segmentation.

We want to increase the neural network tool's performance at segmenting cells with cell perimeter masks that have high Intersection over Union (IoU) with the ground truth masks.

The cell images have dimensions (B, L, W, C) = (batch, length, width, channel). To correctly predict masks, the images provided must be in the format of standard ImageData object and must maintain channel dimensions and ordering.

Task prompt The agent is provided with a prompt stating this task is for biological segmentation with the goal of improving segmentation performance using OpenCV functions. The LLM is encouraged to try either novel combinations of preprocessing functions or the hyperparameters within, as well as to build up incrementally more complex functions. The agent is provided with a description of and told to maximize the score: average precision at Intersection over Union (IoU) threshold of 0.5.

Cellpose Cell Segmentation Task Metrics Details

The following metrics are used to evaluate the performance of the pipeline: average_precision.

The average_precision is the average precision score of the pipeline at an Intersection over Union (IoU) threshold of 0.5.

Our ultimate goal is to increase the average_precision as much as possible (0.95 is the target).

Data Split For this case study, we curated a publicly available and reconstructable subset from the reported Cellpose3 dataset, including test sets from the Cellpose3 dataset release (68 images), improved TissueNet 1.1 test set (1324 images), Omnipose fluorescent bacterial test set (75 images), and Omnipose phase-contrast bacterial test set (148 images). Datasets involving complex mask corrections were excluded. All constituent datasets (Cellpose, Omnipose bacterial fluorescence and phase-contrast, and TissueNet1.1) were randomized and equally split into a validation set and a testing set (for final evaluation). We then randomly sampled 100 image segmentation mask pairs to use for agentic optimization. We release the code to generate these splits, as well as the remaining $\sim$ 700 validation images for future scaling experiments. Evaluation always occurs on the entire test set. Images were standardized to float32 with pixel intensities scaled to $[0, 1]$, formatted as three-channel images (nuclear channel in red, cytoplasmic/grayscale in green, blue empty), consistent with Cellpose3 input requirements. Image resolutions varied from 66x58 to 2030x2030. Ground truth consists of instance segmentation masks for all cells. For agentic optimization, we randomly sampled 100 images from the validation set. Final evaluation was conducted on the entire generated test set (807 images).

A.11.4 MedSAM

Data Prompt The agent is given a prompt describing that data features images from the X-ray and dermoscopy modalities. The agent is also told that input images are 3-channel.

Blue boxes mark text deleted for the data prompt ablation. Peach underlined text marks text deleted for the task prompt ablation.

About the Dataset

This is a large-scale medical image segmentation dataset, covering the dermoscopy/xray modality The images have dimensions (H, W, C) = (height, width, channel).

Task Prompt The agent is also given a prompt stating that the task is medical image segmentation, with the goal of improving performance using OpenCV functions. The LLM is encouraged to design new preprocessing methods and instructed to maximize performance based on the sum of the Normalized Surface Dice (NSD) and Dice Similarity Coefficient (DSC) scores.

Medical Segmentation Task Metrics Details

The following metrics are used to evaluate the performance of the pipeline:

- `dsc_metric` is the dice similarity coefficient (DSC) score of the pipeline and is similar to IoU, measuring the overlap between predicted and ground truth masks.
- `nsd_metric` is the normalized surface distance (NSD) score and is more sensitive to distance and boundary calculations.

Data Split We selected a subset of 2D images from the Codabench validation set for each modality (66 images for dermoscopy, 379 images for X-ray), randomly shuffled them with corresponding bounding boxes and segmentation masks, and equally split them into validation and test sets. Images from both modalities were resized to 1024×1024 pixels with three channels to match the MedSAM encoder input. Ground truth included binary segmentation masks of target objects and associated bounding box prompts. We shuffled and split the image-prompt-mask tuples (66 for Dermoscopy, 379 for X-ray) equally into validation and test sets (33-33 for Dermoscopy, 180-179 for X-ray). A smaller sample of 25 images was used for agentic optimization for each modality.

A.12 Computational requirements

A.12.1 Polaris

The full system (including function library, data prompt, task prompt, with GPT-4.1) took 12 hours 27 minutes and 5 seconds for 20 rollouts on 30 vCPUs of a Intel Xeon Platinum 8358 CPU, running serially on 2 Nvidia A10 GPUs.

A.12.2 Cellpose

The full system (including function library, data prompt, task prompt, with GPT-4.1) took 1 hr 42 minutes and 25 seconds for 20 parallel rollouts on a AMD EPYC 7763 64-Core Processor machine, distributed across 8 Nvidia A6000 48 GB GPUs.

A.12.3 MedSAM

The full system (including function library, data prompt, and task prompt, with GPT-4.1) was executed as 20 parallel rollouts on an AMD EPYC 7763 64-Core Processor machine, distributed across 8 NVIDIA A100 80GB SXM4 GPUs.

A.12.4 AutoML

For each task, AutoML was run for 1200 trials on a single Nvidia A100 40GB GPU.

A.13 Primitives Used in Functions

`cv.bilateralFilter(src, d, sigmaColor, sigmaSpace[, dst[, borderType]])` Applies the bilateral filter to an image.

`cv.blur(src, ksize[, dst[, anchor[, borderType]]])` Blurs an image using the normalized box filter.

`cv.boxFilter(src, ddepth, ksize[, dst[, anchor[, normalize[, borderType]]]])` Blurs an image using the box filter.

`cv.dilate(src, kernel[, dst[, anchor[, iterations[, borderType[, borderValue]]]]])` Dilates an image by using a specific structuring element.

`cv.erode(src, kernel[, dst[, anchor[, iterations[, borderType[, borderValue]]]]])` Erodes an image by using a specific structuring element.

`cv.GaussianBlur(src, ksize, sigmaX[, dst[, sigmaY[, borderType[, hint]]]])` Blurs an image using a Gaussian filter.

`cv.getDerivKernels(dx, dy, ksize[, kx[, ky[, normalize[, ktype]]]])` Returns filter coefficients for computing spatial image derivatives.

`cv.getGaborKernel(ksize, sigma, theta, lambd, gamma[, psi[, ktype]])` Returns Gabor filter coefficients.

`cv.getGaussianKernel(ksize, sigma[, ktype])` Returns Gaussian filter coefficients.

`cv.getStructuringElement(shape, ksize[, anchor])` Returns a structuring element of the specified size and shape for morphological operations.

`cv.Laplacian(src, ddepth[, dst[, ksize[, scale[, delta[, borderType]]]]])` Calculates the Laplacian of an image.

`cv.medianBlur(src, ksize[, dst])` Blurs an image using the median filter.

`cv.pyrMeanShiftFiltering(src, sp, sr[, dst[, maxLevel[, termcrit]]])` Performs initial step of meanshift segmentation of an image.

`cv.pyrUp(src[, dst[, dstsize[, borderType]]])` Upsamples an image and then blurs it.

`cv.Scharr(src, ddepth, dx, dy[, dst[, scale[, delta[, borderType]]]])` Calculates the first x- or y- image derivative using Scharr operator.

`cv.sepFilter2D(src, ddepth, kernelX, kernelY[, dst[, anchor[, delta[, borderType]]]])` Applies a separable linear filter to an image.

`cv.Sobel(src, ddepth, dx, dy[, dst[, ksize[, scale[, delta[, borderType]]]]])` Calculates the first, second, third, or mixed image derivatives using an extended Sobel operator.

`cv.morphologyEx(src, op, kernel[, dst[, anchor[, iterations[, borderType[, borderValue]]]]])` Performs advanced morphological transformations.

`cv.spatialGradient(src[, dx[, dy[, ksize[, borderType]]]])` Calculates the first order image derivative in both x and y using a Sobel operator.

`cv.sqrBoxFilter(src, ddepth, ksize[, dst[, anchor[, normalize[, borderType]]]])` Calculates the normalized sum of squares of the pixel values overlapping the filter.

`cv.stackBlur(src, ksize[, dst])` Blurs an image using the stackBlur.

`cv.Canny(image, threshold1, threshold2[, edges[, apertureSize[, L2gradient]]])` Finds edges in an image using the Canny algorithm.

`cv.cornerEigenValsAndVecs(src, blockSize, ksize[, dst[, borderType]])` Calculates eigenvalues and eigenvectors of image blocks for corner detection.

`cv.cornerHarris(src, blockSize, ksize, k[, dst[, borderType]])` Harris corner detector.

`cv.cornerMinEigenVal(src, blockSize[, dst[, ksize[, borderType]]])` Calculates the minimal eigenvalue of gradient matrices for corner detection.

`cv.cornerSubPix(image, corners, winSize, zeroZone, criteria)` Refines the corner locations.

`cv.goodFeaturesToTrack(image, maxCorners, qualityLevel, minDistance[, corners[, mask[, blockSize[, useHarrisDetector[, k]]]]])` Determines strong corners on an image.

`cv.HoughCircles(image, method, dp, minDist[, circles[, param1[, param2[, minRadius[, maxRadius]]]]])` Finds circles in a grayscale image using the Hough transform.

`cv.HoughLines(image, rho, theta, threshold[, lines[, srn[, stn[, min_theta[, max_theta]]]]])` Finds lines in a binary image using the standard Hough transform.

`cv.HoughLinesP(image, rho, theta, threshold[, lines[, minLineLength[, maxLineGap]]])` Finds line segments in a binary image using the probabilistic Hough transform.

`cv.HoughLinesPointSet(point, lines_max, threshold, min_rho, max_rho, rho_step, min_theta, max_theta, theta_step[, lines])` Finds lines in a set of points using the standard Hough transform.

`cv.preCornerDetect(src, ksize[, dst[, borderType]])` Calculates a feature map for corner detection.

`cv.calcBackProject(images, channels, hist, ranges[, backProject[, scale[, uniform]]])` Calculates the back projection of a histogram.

`cv.calcHist(images, channels, mask, histSize, ranges[, hist[, accumulate[, uniform]]])` Calculates a histogram of a set of arrays.

`cv.compareHist(H1, H2, method)` Compares two histograms.

`cv.createCLAHE([clipLimit[, tileGridSize]])` Creates a smart pointer to a cv.CLAHE object and initializes it.

`cv.equalizeHist(src)` Equalizes the histogram of a grayscale image.

`cv.addWeighted(src1, alpha, src2, beta, gamma[, dst[, dtype]])` Calculates the weighted sum of two arrays.

`cv.normalize(src, dst[, alpha[, beta[, norm_type[, dtype[, mask]]]]])` Normalizes the norm or value range of an array.

`cv.adaptiveThreshold(src, maxValue, adaptiveMethod, thresholdType, blockSize, C[, dst])` Applies an adaptive threshold to an array.

`cv.blendLinear(src1, src2, weights1, weights2[, dst])` Performs linear blending of two arrays using specified weights.

`cv.distanceTransform(src, distanceType, maskSize[, dst[, dstType]])` Calculates the distance to the closest zero pixel for each pixel of the source image.

`cv.floodFill(image, seedPoint, newVal[, loDiff[, upDiff[, flags[, mask[, rect]]]]])` Fills a connected component with the given color.

`cv.integral(src[, sum[, sdepth]])` Calculates the integral image.

`cv.integral2(src[, sum[, sqsum[, sdepth[, sqdepth]]]])` Calculates the integral and squared integral images.

`cv.integral3(src[, sum[, sqsum[, tilted[, sdepth[, sqdepth]]]]])` Calculates the integral, squared integral, and tilted integral images.

`cv.threshold(src, thresh, maxval, type[, dst])` Applies a fixed-level threshold to each array element.

`cv.fastNlMeansDenoising(src[, dst[, h[, templateWindowSize[, searchWindowSize]]]])` Perform image denoising using Non-local Means Denoising algorithm.

`cv.fastNlMeansDenoisingColored(src[, dst[, h[, hColor[, templateWindowSize[, searchWindowSize]]]]])` Modification of fastNlMeansDenoising function for colored images.

`cv.cvtColor(src, code[, dst[, dstCn]])` Converts an image from one color space to another.

`cv.merge(mv[, dst])` Creates one multi-channel array out of several single-channel ones.