CoMind: Towards Community-Driven Agents for Machine Learning Engineering

Anonymous Author(s)
Affiliation
Address
email

Abstract

Large language model-based machine learning (ML) agents have shown great promise in automating ML research. However, existing agents typically operate in isolation on a given research problem, without engaging with the broader research community, where human researchers often gain insights and contribute by sharing knowledge. To bridge this gap, we introduce MLE-Live, a live evaluation framework designed to assess an agent's ability to communicate with and leverage collective knowledge from a simulated Kaggle research community. Building on this framework, we propose CoMind, a novel agent that excels at exchanging insights and developing novel solutions within a community context. CoMind achieves state-of-the-art performance on MLE-Live and outperforms 79.2% human competitors on average across four ongoing Kaggle competitions.

1 Introduction

Large language model (LLM)-based agents have shown remarkable potential in automating complex reasoning and decision-making tasks, ranging from software engineering (Jimenez et al., 2023a; Xia et al., 2025) and mathematical problem solving (OpenAI, 2024; Ren et al., 2025; Li et al., 2025) to scientific research exploration (Romera-Paredes et al., 2024; Yamada et al., 2025; Sun et al., 2025; Feng et al., 2025). Among these domains, machine learning engineering (MLE) remains a particularly impactful yet challenging application area, requiring design, implementation, and evaluation of high-performing models across diverse data science tasks.

Recent advances have introduced LLM agents capable of autonomously developing machine learning pipelines for Kaggle-style competitions (Chan et al., 2025). For example, MLAB (Huang et al., 2024) adopts a ReAct-style (Yao et al., 2023) agent for structured decision-making across tasks. AIDE (Jiang et al., 2025) leverages tree-based exploration for improved efficiency, and AutoKaggle (Li et al., 2024) introduces a multi-agent system with skill specialization. These agents have made progress toward end-to-end automation of MLE.

However, existing systems typically operate in isolation, relying solely on internal memory and trial-and-error exploration while ignoring a critical component of real-world scientific practice — community knowledge sharing. In real data science competitions and research workflows, participants frequently learn from public discussions, shared notebooks, and community insights. Such collective knowledge significantly enhances solution quality and innovation. Current agents, due to the inability to engage with this dynamic external context, often converge to repetitive strategies and plateau in performance. Therefore, we are motivated to explore the following critical research question:

*How can we **evaluate** and **design** research agents that utilize collective knowledge?*

To address this question, we introduce **MLE-Live**, a novel live evaluation framework simulating a Kaggle-style research community. Unlike prior benchmarks, MLE-Live includes time-stamped

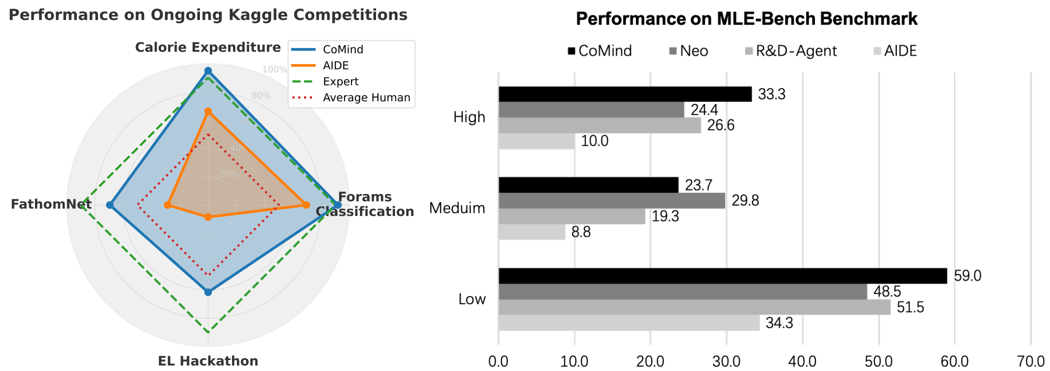

Figure 1: **Left:** CoMind's win rates on four 4 ongoing Kaggle competitions. **Right:** CoMind achieves state-of-the-art performance on the MLE-Bench competitions, measured by *Any Medal* score.

public discussions and shared code artifacts available before competition deadlines – resources that human participants routinely leverage. This setup allows us to rigorously evaluate an agent's ability to use community knowledge in a realistic, temporally grounded setting. In addition, MLE-Live supports both offline evaluation on past competitions and online evaluation on ongoing competitions, enabling comprehensive assessment across static and dynamic scenarios.

Building upon this framework, we propose **CoMind**, a new LLM-based MLE agent that systematically explores diverse ideas, iteratively refines solutions, and selectively incorporates external knowledge. CoMind maintains an evolving idea pool and constructs multiple distinct solution drafts in parallel. It dynamically focuses on one draft at a time, enabling efficient implementation without prompt overflow while preserving technical accuracy. Inspired by human brainstorming, this design balances exploratory breadth with practical depth.

We evaluate CoMind in both previous and ongoing data science competitions. For evaluation on previous competitions, CoMind is tested on the MLE-Live benchmark, which includes 75 past Kaggle competitions on MLE-Bench. CoMind achieves state-of-the-art performance on the leaderboard, significantly outperforming prior agents such as AIDE and R&D-Agent. For evaluation on ongoing competitions, we deploy CoMind on four ongoing Kaggle competitions, where it outperforms 79.2% human competitors on average (Figure 1), demonstrating strong real-world practicality. Human evaluations further confirm that CoMind generates more sophisticated and longer code, reflecting deeper reasoning and better integration of novel insights.

In summary, our contributions are:

- **MLE-Live**: A live evaluation framework simulating community-driven machine learning research, including shared discussions and code for realistic agent benchmarking.
- **CoMind**: A novel LLM-based agent that excels at leveraging collective knowledge and iterative idea exploration, achieving medal-level performance in real Kaggle competitions.
- **Agent design innovations**: We propose a new iterative and parallel exploration mechanism that enables continuous knowledge accumulation, improves diversity, and overcomes LLM context window limitations.

2 Related Work

The rise of large language models (LLMs) has sparked a new wave of research into LLM-driven agents — systems that leverage LLMs' reasoning and language capabilities to autonomously perceive, plan, and act within digital or physical environments. Early works such as ReAct (Yao et al., 2023; Schick et al., 2023; Shen et al., 2023; Hong et al., 2023; Boiko et al., 2023) introduced frameworks that transform LLMs into programmable reasoning engines by interleaving natural language reasoning with tool-use actions. Subsequent studies have extended these agents to various domains, including computer usage (Xie et al., 2024; Zhou et al., 2024) and software development (Wang et al., 2025; Jimenez et al., 2023b).

In parallel, the field of automated machine learning (AutoML) aims to reduce human involvement in building ML pipelines by automating tasks such as model selection, hyperparameter tuning, and architecture search. Early systems like Auto-WEKA (Thornton et al., 2013), HyperBand (Li et al., 2018) and Auto-sklearn (Feurer et al., 2022) used early stopping and Bayesian optimization to search over pipeline configurations, while methods like DARTS (Liu et al., 2019) expanded automation to neural architectures. More recent frameworks such as AutoGluon (Erickson et al., 2020) and FLAML (Wang et al., 2021) emphasize efficiency and ease of use.

Building on these developments, recent efforts have applied LLM-based agents to machine learning engineering (MLE) tasks (Hollmann et al., 2023; Guo et al., 2024; Li et al., 2024; Grosnit et al., 2024; Hong et al., 2024; Chi et al., 2024; Trirat et al., 2024; Huang et al., 2024). However, most evaluations remain constrained to closed-world settings with predefined search spaces, offering limited insight into how these agents perform in open-ended or collaborative ML environments. While some agents (Guo et al., 2024; AI-Researcher, 2025) incorporate basic retrieval tools, these are typically based on simple semantic matching, and robust evaluation methodologies remain underdeveloped.

Meanwhile, several benchmarks have been proposed to evaluate machine learning (ML) engineering capabilities. MLPerf (Mattson et al., 2020) assesses system-level performance, including training speed and energy efficiency. To evaluate end-to-end ML workflows, MLAB (Huang et al., 2024) tests the capabilities of LLM-based agents across 13 ML tasks. MLE-Bench (Chan et al., 2025) and DSBench (Jing et al., 2025) further extends to about 75 Kaggle competitions covering tasks such as preprocessing, modeling, and evaluation. However, these benchmarks typically evaluate agents in isolation, overlooking the collaborative dynamics of real-world ML development. In contrast, our work introduces a framework that simulates community-driven settings, enabling evaluation of agents' ability to engage with and benefit from shared knowledge — while ensuring that resource access remains fair and realistic.

3 MLE-Live

Existing benchmarks typically evaluate ML agents in static, isolated settings, where agents work independently without interacting with other participants or leveraging community insights. This contrasts sharply with real-world machine learning workflows, particularly on platforms like Kaggle, where collaboration, public sharing, and discussion are essential drivers of innovation.

To enable more realistic and comprehensive evaluation of agents in community-driven research settings, we introduce MLE-Live, a live evaluation framework built upon Kaggle-style competitions. MLE-Live extends the MLE-Bench benchmark (Chan et al., 2025) by incorporating simulated community interactions that mirror how human participants access and utilize shared knowledge during competitions.

Each competition in MLE-Live includes: (i) Task description: Background information, task specifications, evaluation metrics, and dataset structure scraped directly from the original Kaggle competition pages. (ii) Competition dataset: A cleaned train-test split, constructed when necessary to account for unavailable or partial test data after competition closure. (iii) Submission grader: An evaluation script that mimics Kaggle's scoring mechanism. (iv) Leaderboard: A ranking system to reflect solution quality and progress.

Simulated Community Environment. Beyond the standard competition setup, MLE-Live introduces a community simulation system that mimics the collaborative ecosystem of real Kaggle competitions. Specifically, we collect and curate:

- **Shared discussions**, including strategy brainstorming, model diagnostics.
- **Public kernels**, including end-to-end solutions and code snippets which were posted *before* the official competition deadline.

These artifacts reflect the auxiliary resources human participants would naturally reference, making MLE-Live a richer and more authentic testbed for ML agents.

In total, we collected 12,951 discussions and 15,733 kernels across 75 Kaggle competitions from MLE-Bench.

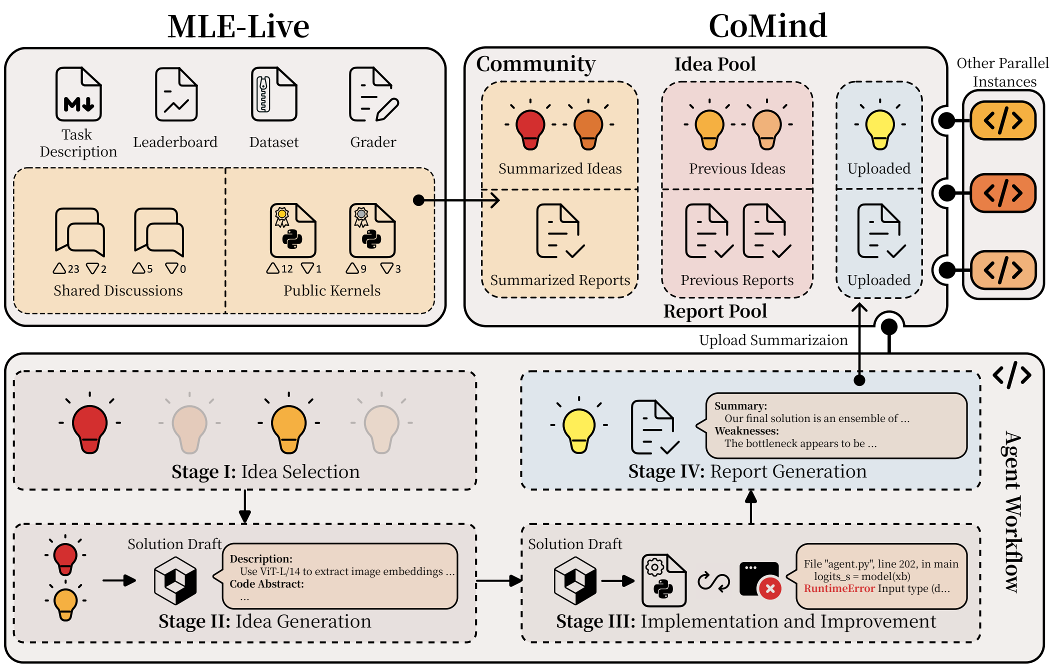

Figure 2: Overview of MLE-Live and the workflow of CoMind. CoMind simulates a community and operates iteratively through four stages: *Idea Selection*, *Idea Generation*, *Implementation and Improvement*, and *Report Generation*. Multiple agents on the same task share the community knowledge base, new reports will be added and visible to others in subsequent iterations.

Metadata and Quality Signals. Each resource in MLE-Live is augmented with critical metadata to help agents and evaluators prioritize relevant, high-quality content:

- **Vote count:** Community preference indicator; highly voted items often contain well-structured insights.
- **Public score:** Automatically computed by Kaggle on the public test split, indicating kernel performance.
- **Author tier:** A qualitative marker of the contributor's expertise, ranging from Novice to Grandmaster.

Importantly, all included content was published before competition deadlines, ensuring a faithful simulation of real-time knowledge sharing without post-hoc leakage. This design makes MLE-Live a controlled yet realistic benchmark to assess how well agents can leverage collective intelligence during the research process.

4 CoMind

We introduce **CoMind**, a community-augmented large language model (LLM) agent designed to automate machine learning (ML) engineering in an iterative, collaborative setting. Figure 2 is a overview of CoMind workflows. Inspired by the workflow of human practitioners on platforms like Kaggle, CoMind operates in a loop that mirrors how experts read community posts, form new ideas, experiment, and share results. The system operates in iterative cycles, each consisting of four stages: *Idea Selection*, *Idea Generation*, *Implementation and Improvement*, and *Report Generation*. To support cumulative progress, CoMind maintains two central repositories: an *idea pool* containing abstracted insights derived from community content and prior iterations, and a *report pool* containing finalized solution reports with associated code, evaluations, and analyses. Additionally, we extend the setting to support multi-agent collaboration through shared insight exchange. These components facilitate both intra-agent memory and inter-agent communication in the multi-agent deployments.

4.1 Agent Workflow

Stage I: Idea Selection. At the beginning of each iteration, CoMind accesses the idea pool, which contains curated concepts and strategies distilled from previous solutions, public kernels and forum discussions. By utilizing the report pool as a guidance of performance and relevance assessment of entries in the idea pool, CoMind ranks and filters these entries to identify a subset of ideas most promising for the current task. This process mimics how human participants explore collective wisdom before forming new hypotheses and experimenting.

Stage II: Idea Generation. Based on the selected ideas and additional context from the report pool, which contains detailed descriptions of previous solution implementations and their empirical performance, CoMind generates a high-level *solution draft*. This draft synthesizes new strategies by recombining or extending the selected ideas. Importantly, it is designed to avoid simple replication, thereby ensuring conceptual diversity and promoting exploratory breadth. This stage reflects the human capacity for abstraction and innovation, where participants generalize from past work to create novel solution blueprints.

Stage III: Implementation and Improvement. Based on the generated solution draft, CoMind initiates a ReAct-style loop to implement, validate and refine the pipeline. In this stage, the CoMind iteratively issues code snippets, executes them within a constrained and isolated runtime environment, observes feedback (e.g., validation metrics, error logs), and updates its implementation accordingly. This loop continues for a fixed time budget, allowing CoMind to incrementally debug and optimize its solution through trial and error. Notably, the agent's contextual input during this stage is deliberately restricted to include only the problem statement, the specific solution draft, and execution feedback, excluding direct access to the broader ideal pool and report pool. This ensures that the CoMind develops the solution path independently, preserving experimental modularity while preventing the underlying explosion of context length.

Stage IV: Report Generation. Upon convergence or budget exhaustion, CoMind compiles a comprehensive report for the solution draft, which consist of: (1) a clear description of the proposed method; (2) an analysis of each major component; (3) quantitative performance results; and (4) an assessment of limitations and future directions. The resulting report is then posted back to the simulated community by added to the report pool, making it available to other agents in future iterations. This mirrors how real users document and share their final solutions.

4.2 Parallel Agents with Shared Insight

Beyond a single-agent loop, CoMind also supports a collaborative multi-agent setting. Multiple agents operate in parallel on the same task, each with access to the shared community knowledge base. As agents generate new reports, these are added to the pool and can be read by others in subsequent iterations. This allows agents to build upon each other's ideas, fostering community-driven exploration and collective improvement.

5 Benchmark Evaluation

5.1 Setup

Task Selection. Based on MLE-Live evaluation framework, we evaluate our agent on 75 Kaggle competitions on MLE-Bench. Using the MLE-Live framework, CoMind has access to shared discussions and public kernels published on the competition websites before the competition deadline.

To validate CoMind under realistic conditions, we further evaluate CoMind on four ongoing Kaggle competitions: `el-hackathon-2025`, `fathomnet-2025`, `playground-series-s5e5` and `forams-classification-2025`. These competitions span diverse domains, including tabular learning, image classification and 3D object classification. Rather than approximating the official scoring locally, we directly submit CoMind's generated `submission.csv` files to the Kaggle platform, so that all reported ranks reflect genuine, live leaderboard positions. Notably, `fathomnet-2025` is part of the Fine-Grained Visual Categorization (FGVC12) workshop at the CVPR Conference. Unlike typical Kaggle competitions, a panel of experts will review the top entries based not only on scores

Table 1: **Any Medal (%) scores on MLE-Bench competitions.** Best results in each column are highlighted in bold. Baseline results are from the official leaderboard.

Agent	Low (%)	Medium (%)	High (%)	All (%)
CoMind o4-mini	**59.09**	23.68	**33.33**	**36.00**
Neo multi-agent	48.48	**29.82**	24.44	34.22
R&D-Agent o3 + GPT-4.1	51.52	19.30	26.67	30.22
ML-Master deepseek-r1	48.50	20.20	24.40	29.30
R&D-Agent o1-preview	48.18	8.95	18.67	22.40
AIDE o1-preview	34.30	8.80	10.00	16.90
AIDE gpt-4o	19.00	3.20	5.60	8.60
AIDE claude-3-5-sonnet	19.40	2.60	2.30	7.50
OpenHands gpt-4o	11.50	2.20	1.90	5.10
AIDE llama-3.1-405b-instruct	8.30	1.20	0.00	3.10
MLAB gpt-4o	4.20	0.00	0.00	1.30

but also on the methodological descriptions. Although the competition offers no monetary prize, it serves as a high-profile venue for academic and practical contributions to marine biodiversity research.

Implementation Details. CoMind employs `o4-mini-2025-04-16` (OpenAI, 2025) as its backend LLM. We limit the hardware constraint of each run to 32 vCPUs and a single A6000 GPU. Each competition is evaluated in separate containers with a maximum of 24 hours to produce the final submission file. Every single code execution session is limited to 5 hour. The *Implementation and Improvement* stage of CoMind is limited to a maximum of 20 steps. The number of parallel agents is set to 4.

During code generation, agents are provided with the test set inputs (without labels) and prompted to generate a `submission.csv` file. The submission is then evaluated by a grader that compares the predicted labels with the ground truth. Following the setting of MLE-Bench, to avoid potential overfitting, test set labels and the competition leaderboard are strictly withheld from the agent's accessible environment. Instead, each agent must rely solely on a self-constructed "runtime test set", a held-out split from the original training data, for code evaluation and performance estimation.

Metrics. Following the evaluation metrics in MLE-Bench, we measure the performance of CoMind by **Any Medal**, the percentage of competitions where the agent earns a gold, silver, or bronze medal.

Baselines. We compare CoMind against the MLE-Bench leaderboard[1] including open-sourced systems like **R&D-Agent** (Yang et al., 2025), a dual-agent framework (Researcher/Developer) that explores multiple solution branches and merges promising ideas into improved pipelines; **ML-Master** (Liu et al., 2025), which integrates exploration and reasoning via a selectively scoped memory that aggregates insights from parallel trajectories; **AIDE** (Jiang et al., 2025), a purpose-built tree-search scaffold that iteratively drafts, debugs, and benchmarks code for Kaggle-style tasks; **OpenHands** (Wang et al., 2025), a general-purpose CodeAct-based scaffold that executes code and calls tools in a sandboxed environment; **MLAB** (Huang et al., 2024), referring to the ResearchAgent scaffold from MLAgentBench, a general tool-calling/plan–act baseline; and **Neo** (`https://heyneo.so/`), a close-sourced multi-agent system for autonomous ML engineering.

5.2 Results

Table 1 compares CoMind with baseline methods on 75 MLE-Bench competitions. CoMind achieves state-of-the-art performance with an *Any Medal* rate of 36.00%, significantly outperforming open-source competitors such as R&D-Agent (submitted on 2025-08-15) and surpassing the closed-source multi-agent system Neo. Appendix C provides a detailed case study on `denoising-dirty-documents`.

[1] `https://github.com/openai/mle-bench`

Table 2: **Authentic scores and top-percentile ranks of CoMind and AIDE on ongoing Kaggle competitions.** "Higher better" marks whether a larger score is better for that competition. "Top %" is the percentile rank on competition leaderboard (lower is better). In `playground-series-s5e5` and `fathomnet-2025`, lower scores are better.

Competition	Score higher better	CoMind		AIDE	
		Score	Top %	Score	Top %
`playground-series-s5e5`	×	**0.5673**	5.1%	0.5772	33.8%
`forams-classification-2025`	✓	**0.7645**	8.3%	0.6041	30.6%
`el-hackathon-2025`	✓	**0.5837**	38.4%	0.1140	91.5%
`fathomnet-2025`	×	**2.81**	30.6%	3.71	71.4%

On the four evaluated ongoing competitions CoMind's standings are: `playground-series-s5e5` (#120 out of 2,338); `forams-classification-2025` (#4 out of 48); `el-hackathon-2025` (#128 out of 333); `fathomnet-2025` (#15 out of 47). Details including authentic scores and win rates per task are provided in Table 2. These authentic results demonstrate CoMind's capability to tackle a variety of problem domains and achieve competitive performance in live, evolving ML workflows. In particular, our success in the CVPR-affiliated `fathomnet-2025` challenge highlights CoMind's potential to contribute meaningfully not only to industrial applications but also to scientific and interdisciplinary research communities.

6 Ablation Study

6.1 Setup

Task Selection. To evaluate the impact of introducing public resources, we conducted an ablation study on 20 competitions from MLE-Bench-Lite based on MLE-Live. These tasks span across various categories, including image classification/generation, text classification/generation, image regression, audio classification, and tabular analysis.

Baselines. We compared CoMind against the following baselines. For consistency, all baselines use the same backend model as CoMind:

- **AIDE+Code.** To enable the use of publicly available code (e.g., Kaggle kernels), we extend AIDE with access to one public kernel per draft node—selected by highest community votes. This version, AIDE+Code, augments the prompt with both the task description and the selected kernel alongside the tree summarization.
- **AIDE+RAG.** We further equip AIDE with a retrieval-augmented generation (RAG) mechanism. Before generating code, the agent retrieves the titles of the top 10 voted discussions and kernels. The LLM selects the most relevant ones, receives a summarization, and then proposes its plan and implementation. For debugging or refinement, it can optionally re-query documents. Retrieval is based on cosine similarity between query and candidate document embeddings, using Multilingual E5 Text Embeddings (Wang et al., 2024).
- **CoMind w/o $\mathcal{R}$.** In this variant, CoMind operates without access to any external community resources. It starts with empty idea and report pools and relies solely on its own generation history to propose candidate ideas and assemble solution drafts.

Metrics. Following the evaluation metrics in prior research (Chan et al., 2025), the relative capability of generating high-quality solution compared with human is measured by:

- **Above Median**: Indicates whether the submission outperforms at least 50% of competitors on the leaderboard.
- **Win Rate**: The percentage of competitors whose final scores are lower than the agent's score. If the agent fails to produce a valid submission, the Win Rate is 0.
- **Medals**: Medals are assigned based on the agent's score relative to Kaggle leaderboard thresholds for gold, silver, and bronze medals.
- **Any Medal**: The percentage of competitions in which the agent earns any medal.

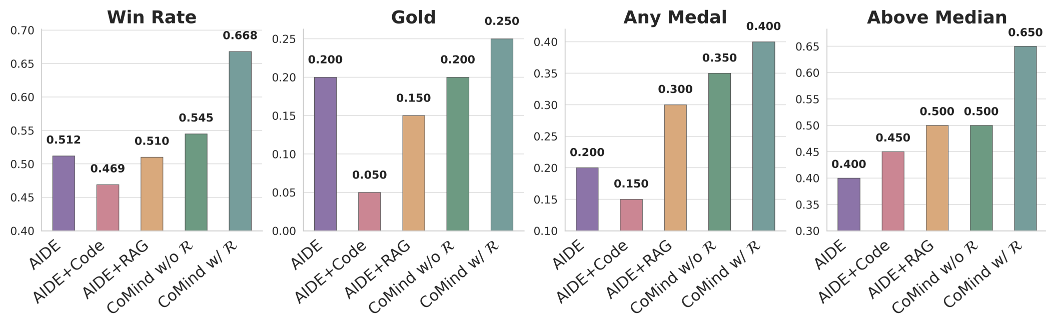

Figure 3: **Performance of CoMind and other baselines on 20 competitions from MLE-Bench-Lite.** *Valid Submission* is the ratio of submissions meeting format requirements and validation criteria. *Win Rate* is the percentage of human competitors outperformed by the agent. *Any Medal*, is the proportion of competitions where the agent earned Gold, Silver or Bronze medals. *Above Median* is the fraction of competitions where the agent's score strictly exceeded the median human competitor.

Table 3: **Average win rate of CoMind and other baselines across task categories on 20 competitions from MLE-Bench-Lite.** *# of Tasks* refers to the number of competitions in the corresponding category. Notably, CoMind demonstrated superior performance in Image Classification, Text Classification, Audio Classification and Image To Image.

Category	# of Tasks	CoMind	AIDE+Code	AIDE+RAG	AIDE
Image Classification	8	**0.597**	0.459	0.434	0.525
Text Classification	3	**0.740**	0.157	0.338	0.61
Audio Classification	1	**0.901**	0.272	0.259	0.271
Seq2Seq	2	0.408	0.503	**0.550**	0.228
Tabular	4	0.664	0.673	**0.688**	0.483
Image To Image	1	**0.988**	0.932	0.617	0.568
Image Regression	1	**0.992**	0.342	**0.992**	**0.992**
All	20	**0.668**	0.469	0.510	0.512

Implementation Setup. All agents use `o4-mini-2025-04-16` as their backend. Based on the settings of our main experiment, the hardware constraint is further limited to 4 vCPUs and 5 hours per competition. Each execution session is limited to 1 hour.

6.2 Results

Figure 3 shows the results. Our key findings are as follows: (i) CoMind consistently outperforms all baselines across every metric. (ii) Among the AIDE variants, AIDE+RAG outperforms AIDE+Code, and both surpass the original AIDE on most metrics, demonstrating the benefits of integrating community knowledge. CoMind further exceeds these approaches, highlighting the effectiveness of its deeper and more strategic community-aware exploration. (iii) Removing CoMind's resource access causes a significant drop in valid submission rates and other metrics, showing that strategic access to public resources helps CoMind balance extending established methods for reliability with exploring novel approaches.

7 Analytical Experiments

For analytical experiments, we adopt the same setup as the ablation study and evaluate model performance across multiple dimensions, including task categories, win rate over time, and code complexity.

Task Categories Table 3 reports the average ranks across seven task categories. CoMind outperforms all baselines in Image Classification, Text Classification, Audio Classification, and Image-to-Image tasks, highlighting its strong adaptability. We manually inspect the tasks where CoMind

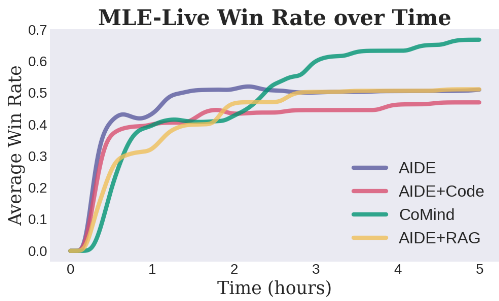

Figure 4: **Average win rate of CoMind and other baselines over time.** AIDE, AIDE+Code, and AIDE+RAG rose rapidly but plateaued, while CoMind continued improving and eventually outperformed them.

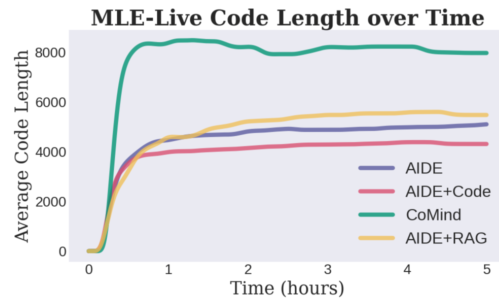

Figure 5: **Average code length (character count) of valid solutions over time.** CoMind maintained a substantially longer code length, suggesting more complex logic and richer optimization techniques.

underperformed and find that the issues are often related to the use of large models or datasets. For example, in Seq2Seq tasks, CoMind explores complex fine-tuning strategies for large language models which often fail to complete within the one-hour runtime constraint.

Win Rate Over Time Figure 4 shows the evolution of average win rate over time. AIDE quickly produces concise, functional solutions, leading to a rapid rise in performance during the first hour. In contrast, CoMind spends more time on debugging and exploration early on, resulting in a slower initial improvement. However, after the first two hours, AIDE's performance plateaus, while CoMind continues to improve through iterative refinement and deeper exploration, ultimately surpassing AIDE and achieving higher-quality solutions.

Code Complexity Regarding code complexity, Figure 5 illustrates the average code length during the entire competition. CoMind consistently generates significantly longer and more complex code, while other baselines begin with simpler implementations and introduce only incremental modifications. Appendix A offers a comparative analysis across code complexity metrics and task categories. Notably, CoMind's solutions for Image Regression and Audio Classification are nearly twice as long as those of other baselines. Additionally, solutions from CoMind are, on average, 55.4% longer than those produced by AIDE.

8 Conclusion

We introduce **MLE-Live**, a new framework for evaluating machine learning agents in realistic, community-driven environments. By simulating the collaborative dynamics of Kaggle competitions with shared discussions and public code, MLE-Live enables a more faithful assessment of agents' ability to leverage collective knowledge. Building upon this framework, we propose **CoMind**, an LLM-based agent that iteratively selects, synthesizes, and implements ideas using both internal reasoning and external insights. CoMind consistently outperforms prior methods on MLE-Live benchmark and four ongoing Kaggle competitions, demonstrating the value of community awareness and iterative exploration in research automation. In addition, MLE-Live lays the groundwork for future studies in collaborative AI systems where agents not only learn from data, but also from each other.

Limitations and Future Work. Currently, CoMind supports only report-level interactions. Expanding the agent's action space to include commenting, question-asking, or sharing datasets and models is a promising next step. In addition, while our current experiments focus on Kaggle-style ML tasks, the MLE-Live framework can be extended to broader domains, such as scientific discovery, open-ended coding, or robotics, enabling research agents to contribute meaningfully across diverse fields.

References

AI-Researcher (2025). Ai-researcher: Fully-automated scientific discovery with llm agents. Accessed: 2025-05-15.

Boiko, D. A., MacKnight, R., Kline, B., and Gomes, G. (2023). Autonomous chemical research with large language models. *Nature*, 624:570 – 578.

Chan, J. S., Chowdhury, N., Jaffe, O., Aung, J., Sherburn, D., Mays, E., Starace, G., Liu, K., Maksin, L., Patwardhan, T., Madry, A., and Weng, L. (2025). MLE-bench: Evaluating machine learning agents on machine learning engineering. In *The Thirteenth International Conference on Learning Representations*.

Chi, Y., Lin, Y., Hong, S., Pan, D., Fei, Y., Mei, G., Liu, B., Pang, T., Kwok, J., Zhang, C., Liu, B., and Wu, C. (2024). Sela: Tree-search enhanced llm agents for automated machine learning. *ArXiv*, abs/2410.17238.

Erickson, N., Mueller, J., Shirkov, A., Zhang, H., Larroy, P., Li, M., and Smola, A. (2020). Autogluon-tabular: Robust and accurate automl for structured data. *arXiv preprint arXiv:2003.06505*.

Feng, S., Sun, W., Li, S., Talwalkar, A., and Yang, Y. (2025). A comprehensive evaluation of contemporary ml-based solvers for combinatorial optimization. *ArXiv*, abs/2505.16952.

Feurer, M., Eggensperger, K., Falkner, S., Lindauer, M., and Hutter, F. (2022). Auto-sklearn 2.0: Hands-free automl via meta-learning. *Journal of Machine Learning Research*, 23(261):1–61.

Grosnit, A., Maraval, A. M., Doran, J., Paolo, G., Thomas, A., Beevi, R. S. H. N., Gonzalez, J., Khandelwal, K., Iacobacci, I., Benechehab, A., Cherkaoui, H., Hili, Y. A. E., Shao, K., Hao, J., Yao, J., Kégl, B., Bou-Ammar, H., and Wang, J. (2024). Large language models orchestrating structured reasoning achieve kaggle grandmaster level. *ArXiv*, abs/2411.03562.

Guo, S., Deng, C., Wen, Y., Chen, H., Chang, Y., and Wang, J. (2024). Ds-agent: Automated data science by empowering large language models with case-based reasoning. *ArXiv*, abs/2402.17453.

Hollmann, N., Müller, S. G., and Hutter, F. (2023). Large language models for automated data science: Introducing caafe for context-aware automated feature engineering. In *Neural Information Processing Systems*.

Hong, S., Lin, Y., Liu, B., Wu, B., Li, D., Chen, J., Zhang, J., Wang, J., Zhang, L., Zhuge, M., Guo, T., Zhou, T., Tao, W., Wang, W., Tang, X., Lu, X., Liang, X., Fei, Y., Cheng, Y., Gou, Z., Xu, Z., Wu, C., Zhang, L., Yang, M., and Zheng, X. (2024). Data interpreter: An llm agent for data science. *ArXiv*, abs/2402.18679.

Hong, S., Zheng, X., Chen, J. P., Cheng, Y., Zhang, C., Wang, Z., Yau, S. K. S., Lin, Z. H., Zhou, L., Ran, C., Xiao, L., and Wu, C. (2023). Metagpt: Meta programming for multi-agent collaborative framework. *ArXiv*, abs/2308.00352.

Huang, Q., Vora, J., Liang, P., and Leskovec, J. (2024). MLAgentbench: Evaluating language agents on machine learning experimentation. In *Forty-first International Conference on Machine Learning*.

Jiang, Z., Schmidt, D., Srikanth, D., Xu, D., Kaplan, I., Jacenko, D., and Wu, Y. (2025). Aide: Ai-driven exploration in the space of code.

Jimenez, C. E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., and Narasimhan, K. (2023a). Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*.

Jimenez, C. E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., and Narasimhan, K. (2023b). Swe-bench: Can language models resolve real-world github issues? *ArXiv*, abs/2310.06770.

Jing, L., Huang, Z., Wang, X., Yao, W., Yu, W., Ma, K., Zhang, H., Du, X., and Yu, D. (2025). DSBench: How far are data science agents from becoming data science experts? In *The Thirteenth International Conference on Learning Representations*.

Li, L., Jamieson, K., DeSalvo, G., Rostamizadeh, A., and Talwalkar, A. (2018). Hyperband: A novel bandit-based approach to hyperparameter optimization. *Journal of Machine Learning Research*, 18(185):1–52.

Li, S., Marwah, T., Shen, J., Sun, W., Risteski, A., Yang, Y., and Talwalkar, A. (2025). Codepde: An inference framework for llm-driven pde solver generation. *arXiv preprint arXiv:2505.08783*.

Li, Z., Zang, Q., Ma, D., Guo, J., Zheng, T., Liu, M., Niu, X., Wang, Y., Yang, J., Liu, J., et al. (2024). Autokaggle: A multi-agent framework for autonomous data science competitions. *arXiv preprint arXiv:2410.20424*.

Liu, H., Simonyan, K., and Yang, Y. (2019). DARTS: Differentiable architecture search. In *International Conference on Learning Representations*.

Liu, Z., Cai, Y., Zhu, X., Zheng, Y., Chen, R., Wen, Y., Wang, Y., Weinan, E., and Chen, S. (2025). Ml-master: Towards ai-for-ai via integration of exploration and reasoning. *ArXiv*, abs/2506.16499.

Mattson, P., Cheng, C., Diamos, G., Coleman, C., Micikevicius, P., Patterson, D., Tang, H., Wei, G.-Y., Bailis, P., Bittorf, V., et al. (2020). Mlperf training benchmark. *Proceedings of Machine Learning and Systems*, 2:336–349.

OpenAI (2024). Learning to reason with llms.

OpenAI (2025). Introducing openai o3 and o4-mini.

Ren, Z. Z., Shao, Z., Song, J., Xin, H., Wang, H., Zhao, W., Zhang, L., Fu, Z., Zhu, Q., Yang, D., Wu, Z. F., Gou, Z., Ma, S., Tang, H., Liu, Y., Gao, W., Guo, D., and Ruan, C. (2025). Deepseek-prover-v2: Advancing formal mathematical reasoning via reinforcement learning for subgoal decomposition.

Romera-Paredes, B., Barekatain, M., Novikov, A., Balog, M., Kumar, M. P., Dupont, E., Ruiz, F. J., Ellenberg, J. S., Wang, P., Fawzi, O., et al. (2024). Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475.

Schick, T., Dwivedi-Yu, J., Dessì, R., Raileanu, R., Lomeli, M., Zettlemoyer, L., Cancedda, N., and Scialom, T. (2023). Toolformer: Language models can teach themselves to use tools.

Shen, Y., Song, K., Tan, X., Li, D., Lu, W., and Zhuang, Y. (2023). Hugginggpt: Solving ai tasks with chatgpt and its friends in hugging face.

Sun, W., Feng, S., Li, S., and Yang, Y. (2025). Co-bench: Benchmarking language model agents in algorithm search for combinatorial optimization. *ArXiv*, abs/2504.04310.

Thornton, C., Hutter, F., Hoos, H. H., and Leyton-Brown, K. (2013). Auto-weka: Combined selection and hyperparameter optimization of classification algorithms. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 847–855.

Trirat, P., Jeong, W., and Hwang, S. J. (2024). Automl-agent: A multi-agent llm framework for full-pipeline automl. *ArXiv*, abs/2410.02958.

Wang, C., Wu, Q., Weimer, M., and Zhu, E. (2021). Flaml: A fast and lightweight automl library. *Proceedings of Machine Learning and Systems*, 3:434–447.

Wang, L., Yang, N., Huang, X., Yang, L., Majumder, R., and Wei, F. (2024). Multilingual e5 text embeddings: A technical report. *arXiv preprint arXiv:2402.05672*.

Wang, X., Li, B., Song, Y., Xu, F. F., Tang, X., Zhuge, M., Pan, J., Song, Y., Li, B., Singh, J., Tran, H. H., Li, F., Ma, R., Zheng, M., Qian, B., Shao, Y., Muennighoff, N., Zhang, Y., Hui, B., Lin, J., Brennan, R., Peng, H., Ji, H., and Neubig, G. (2025). Openhands: An open platform for AI software developers as generalist agents. In *The Thirteenth International Conference on Learning Representations*.

Xia, C. S., Deng, Y., Dunn, S., and Zhang, L. (2025). Demystifying llm-based software engineering agents. *Proceedings of the ACM on Software Engineering*, 2(FSE):801–824.

Xie, T., Zhang, D., Chen, J., Li, X., Zhao, S., Cao, R., Hua, T. J., Cheng, Z., Shin, D., Lei, F., Liu, Y., Xu, Y., Zhou, S., Savarese, S., Xiong, C., Zhong, V., and Yu, T. (2024). OSWorld: Benchmarking multimodal agents for open-ended tasks in real computer environments. In *The Thirty-eight Conference on Neural Information Processing Systems Datasets and Benchmarks Track*.

Yamada, Y., Lange, R. T., Lu, C., Hu, S., Lu, C., Foerster, J., Clune, J., and Ha, D. (2025). The ai scientist-v2: Workshop-level automated scientific discovery via agentic tree search. *arXiv preprint arXiv:2504.08066*.

Yang, X., Yang, X., Fang, S., Xian, B., Li, Y., Wang, J., Xu, M., Pan, H., Hong, X., Liu, W., Shen, Y., Chen, W., and Bian, J. (2025). R&d-agent: Automating data-driven ai solution building through llm-powered automated research, development, and evolution. *ArXiv*, abs/2505.14738.

Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K. R., and Cao, Y. (2023). React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations*.

Zhou, S., Xu, F. F., Zhu, H., Zhou, X., Lo, R., Sridhar, A., Cheng, X., Ou, T., Bisk, Y., Fried, D., Alon, U., and Neubig, G. (2024). Webarena: A realistic web environment for building autonomous agents. In *The Twelfth International Conference on Learning Representations*.

A Additional Analysis on Code Complexity

In this section, we provide a comprehensive analysis of the generated code using a broad set of software complexity and quality metrics, beyond mere line counts. Specifically, we report the following indicators: **Cyclomatic Complexity (CC)**, **Pylint score**, **Halstead Metrics**: Volume, Difficulty, Effort, **Source Lines of Code (SLOC)**, **Number of Comment Lines** and **Code Length**.

Table 4: Code Complexity and Quality Metrics by Task Category.

Category	Metric	CoMind	AIDE	AIDE+RAG	AIDE+Code
Image Classification	CC	1.68	1.59	1.93	1.29
	Pylint Score	7.43	9.06	8.90	8.92
	Volume	330.88	143.26	84.20	175.88
	Difficulty	4.95	2.90	2.32	2.59
	Effort	1960.22	507.06	286.31	725.59
	SLOC	198.25	133.50	120.88	115.71
	Comment Lines	15.62	12.88	13.75	14.43
	Code Length	7638.40	4624.30	4701.30	5192.10
Text Classification	CC	3.58	4.28	2.00	0.00
	Pylint Score	8.82	9.09	8.89	9.26
	Volume	286.38	384.07	47.68	29.25
	Difficulty	3.76	3.94	1.25	1.31
	Effort	1183.11	2332.22	61.56	35.16
	SLOC	181.67	133.00	141.00	69.50
	Comment Lines	14.67	15.33	14.00	13.50
	Code Length	6974.70	3094.50	5920.50	5629.30
Audio Classification	CC	2.00	0.00	0.00	0.00
	Pylint Score	7.92	9.11	9.49	8.86
	Volume	718.63	244.20	115.95	227.48
	Difficulty	7.39	6.46	3.19	6.38
	Effort	5308.07	1577.11	369.58	1451.30
	SLOC	256.00	82.00	92.00	72.00
	Comment Lines	20.00	11.00	16.00	16.00
	Code Length	9449.00	3508.00	4151.00	3352.00
Seq2Seq	CC	4.38	2.25	22.33	15.75
	Pylint Score	8.58	9.04	9.14	8.51
	Volume	492.55	52.33	390.46	324.00
	Difficulty	3.87	2.14	5.26	3.68
	Effort	1935.02	140.58	2083.84	1686.74
	SLOC	184.50	63.50	222.50	147.50
	Comment Lines	22.50	13.00	23.00	19.50
	Code Length	6925.50	5649.50	8357.50	2728.50
Tabular	CC	2.78	1.62	2.38	0.25
	Pylint Score	8.65	8.96	8.87	9.31
	Volume	1264.61	856.12	815.29	435.46
	Difficulty	7.37	4.83	6.05	3.69
	Effort	10 808.93	6163.62	5564.22	2001.06
	SLOC	218.75	139.75	147.50	93.50
	Comment Lines	18.25	14.75	15.25	10.50
	Code Length	8570.00	3534.00	6064.00	5759.80
Image to Image	CC	1.72	2.00	3.00	1.88
	Pylint Score	8.43	6.25	6.64	7.74
	Volume	1298.11	1481.62	414.59	431.08
	Difficulty	9.68	6.73	3.94	3.79
	Effort	12 565.66	9967.24	1633.22	1631.93
	SLOC	228.00	175.00	121.00	128.00
	Comment Lines	26.00	8.00	23.00	13.00
	Code Length	8800.00	5231.00	4815.00	6671.00
Image Regression	CC	1.68	2.00	2.40	2.00

Category	Metric	CoMind	AIDE	AIDE+RAG	AIDE+Code
	Pylint Score	8.62	8.75	8.80	8.89
	Volume	1310.92	241.08	70.32	72.00
	Difficulty	8.75	3.88	2.18	2.73
	Effort	11 466.58	934.17	153.43	196.36
	SLOC	267.00	145.00	116.00	133.00
	Comment Lines	36.00	15.00	12.00	12.00
	Code Length	10 991.00	4841.00	4655.00	5614.00

B Prompts and Responses for CoMind

This section provides some examples of prompts and responses in CoMind, including **Idea Selection**, **Idea Generation**, **Implementation and Improvement** and **Report Generation**.

B.1 Idea Selection

The idea pool is initialized with curated strategies distilled from public kernels and forum discussions before the first iteration. Since the key innovation of CoMind lies in the simulation of the community, we adopt a relatively simple implementation for the distillation, where CoMind only collects and analyzes top-k voted or ranked (with best public score) kernels and discussions.

Prompt for Strategy Distillation of Public Kernels

Introduction You are an expert machine learning researcher preparing for the Kaggle competition described below.
Task Description <*description of the specified task*>
Goals These are top-ranked public scripts during the competition. Your job is to:

1. Carefully read the following scripts.
2. For each script, if it's self-contained, i.e., including model architecture (if there's a model), training strategies, evaluation, etc., then summarize its pipeline.
3. If the pipeline contains technical details, such as extensive feature engineering, hyperparameter tuning, etc., then list them in full detail.
4. Select a representative code segment for each pipeline. You must include dataset reading / submission generation parts. If task-specific details such as feature engineering are included, the code segment should contain them as well.

Public Kernels <*contents of public kernels*>

Response Template of Strategy Distillation of Public Kernels

Pipelines Description of each strategy, separated by ===SEPARATOR=== mark. For each strategy, follow this format:

- Pipeline: A full detailed description of the pipeline. All input/output format, hyperparameters, training settings, model architectures, feature engineering, validation metric, and any other relevant information should be included. **Do not omit any feature engineering details**.
- Code abstract: A representative code segments that captures the essence (including input/output) and novelty of the pipeline. You **MUST** go through all the publicly available code and **include the parts that generate the submission file**. Contain task-specific engineering details. Mark the remainder as ellipses.

Prompt for Strategy Distillation of Public Discussions

Introduction You are an expert machine learning researcher preparing for the Kaggle competition described below.
Task Description *<description of the specified task>*
Goals These are top-voted public discussions during the competition. Your job is to:
Public Discussions *<contents of public discussions>*

1. Carefully read the following discussions.
2. For each discussion, you should decompose it into critical, novel and inspiring ideas that have potential to win this competition.

Response Template of Strategy Distillation of Public Discussions

Ideas required format: python list of strings, each element is a description of an idea extracted from the discussions. e.g. ['idea 1', 'idea 2'].

Once the idea pool is initialized, CoMind enters the main iteration. CoMinds then ranks and filters all entries in the idea pool, following the prompt below.

Prompt for Idea Filtering and Reconstruction

Introduction You are a machine learning expert. After carefully searching the relevant literature, you have come up with a list of ideas to implement. However, this idea list has some issues:

- Some ideas are too similar and should be merged into one.
- Some ideas are overlapping, you should rephrase and decouple them.
- You should discard ideas that are irrelevant to the final performance, such as error visualization, etc.

You should refer to the Reports section and Public Pipelines section for previous implemented pipelines. Please decompose, merge, and reconstruct the ideas listed below.

Ideas *<entries of the idea pool>*

Reports *<entries of the report pool>*

Public Pipelines *<all public pipelines extracted before>*

Response Template of Idea Filtering and Reconstruction

Ideas required format: Python list of strings, each element is a description of an idea extracted from the discussions. e.g. ['idea 1', 'idea 2'].

B.2 Idea Generation

Based on previous ideas and reports. CoMind will first enrich the idea pool by designing and generating other promising strategies for the competition.

Prompt for CoMind Brainstorm

Introduction You are an expert machine learning researcher preparing for the Kaggle competition described below.
Task Description *<description of the specified task>*
Goals I already have a list of ideas that partially explore how to approach this competition. Your job is to:

1. Think creatively and construct at least **4 alternative and highly novel solution paths** that are likely to perform well, especially if combined with careful experimentation.

2. Each solution path can be a strategy, pipeline, or method that combines multiple techniques. Try to make them as different as possible from the existing "ideas" list.
3. After describing each full solution path, **break it down into individual minimal ideas**-these should be the smallest units of implementation (e.g., "use LightGBM for baseline", "normalize input features", "apply stratified K-fold CV")
4. Ensure these ideas do not substantially duplicate items already in "ideas".
5. Refer to the "Reports" section for the latest updates and suggestions on the ideas and previous pipelines.

Ideas *<entries in the idea pool>*

Reports *<entries in the report pool>*

Public Pipelines *<all public pipelines extracted before>*

Instructions Format your output like this (one line, one idea):

Response Template

<your understanding of the task and explanation of your approaches>
===SOLUTION_PATH_1===
<description of this approach>
- *<minimal idea 1>*
- *<minimal idea 2>*
- *<minimal idea 3>*
- ...
===SOLUTION_PATH_2===
...
===SOLUTION_PATH_3===
...

Be ambitious but realistic - many ideas can later be tested on a small subset of the data. Focus on novelty, diversity, and decomposability. Ready? Start.

After brainstorming completes, CoMind synthesizes existing strategies and ideas into several high-level *solution drafts*.

Prompt for Solution Draft Synthesis

Introduction You are an expert machine learning researcher preparing for the Kaggle competition described below.
Task Description *<description of the specified task>*
Ideas *<entries in the idea pool>*
Reports *<entries in the report pool>*
Public Pipelines *<all public pipelines extracted before>*
Goals

1. Carefully read the reports provided above.
2. Based on the ideas and reports, propose *<num_pipes>* **promising self-contained pipelines** that are likely to perform well.
3. The Public pipelines section contains top-ranked public pipelines during the competition. Use them as reference to polish your pipelines.
4. Each pipeline should not overlap with others. Your proposed pipelines should include **one baseline pipeline that uses well-known methods but is robust and relatively easy to implement**. You should reinforce public pipelines and previous pipelines based on their reports (if provided).
5. Ensure that each pipeline can be trained within 2 hours on a single A6000 with 48GB memory.

6. Read the **submission format** requirements in the task description carefully. The format requirement is possible to be different from the training dataset. **THIS IS EXTREMELY IMPORTANT**. Mention in the pipeline descriptions and be sure to include the code that handles the input and output.
7. DO NOT USE tensorflow, use pytorch instead

Response Template for Solution Draft Synthesis

Submit Pipelines Descriptions and codes of pipelines, separated each pipeline by ===SEPARATOR=== mark. For each pipeline, attach code that captures its essential. **You must include the code in public pipelines that handles input and output, and if there are parts of the public pipelines that are similar to the current pipeline, you should include them as well.**

B.3 Implementation

In this stage, all previously generated solution drafts will be distributed to multiple parallel sub-agents. In each instance, CoMind chats with LLM in multiple turns and initiates a ReAct-style loop.

Prompts for Sub-Agent Implementation

Introduction You're an expert Kaggle competitor tasked with implementing a pipeline into Python code. You can modify the details (training parameters, feature engineering, model selection, etc.), but do not change overall architecture of this pipeline. The goal is to **obtain best score** on this competition.
Task Description *<description of the specified task>*
Pipeline *<description of the solution draft to implement>*
Data Overview *<schema of the input file structure>*
Reminders

1. Read the pipeline and task description carefully.
2. **YOUR CODE MUST PRODUCE SUBMISSON AT** *./working-agent_id/submission.csv*, **THIS IS EXTREMELY IMPORTANT**
3. There is one A6000 gpu available for you, **maximize your use of computing resources**. You can use large batchsizes.
4. All the provided input data are **stored in** *./input* **directory**.
5. You can use the *./working-agent_id* directory to store any temporary files that your code needs to create.
6. Include at least one comment explaining your code. **NO PARTS OF THE CODE SHOULD BE SKIPPED OR OMITTED**, don't terminate before finishing the script. Even if your proposed code is a minor change, don't omit any sections that overlap with the previous code.
7. **Remember, your ultimate goal is to Obtain best score on this competition.**
8. Your code should **print the value of the evaluation metric computed on a hold-out validation set**.
9. You can use custom evaluation functions during training, but the final metric **MUST FOLLOW THE EVALUATION SECTION IN THE TASK DESCRIPTION** on a validation set. This is important because we will pick your best code based on this metric.
10. We suggest you to test your code at a small scale and print necessary information before utilizing full dataset to get familiar with the data structure and avoid potential format errors.
11. Time limit per run is 1 hour. Your code will be killed if timeout.

12. Begin by summarizing your understanding of the task, and then propose your first code.

Response Format You should follow the following format:

Response Template for Sub-Agent Implementation

objective of this implementation and suggestions for output evaluation
key technical considerations
expected running time (you should ensure that the code will finish within 1 hour)

```
'''python
your code here
'''
```

A Python environment will be setup and execute the code automatically. After the execution, a summary LLM collects all standard output and determines whether the execution runs successfully.

Prompt for Execution Result Summarization

Introduction You are a Kaggle grandmaster attending a competition. You have written code to solve this task and now need to evaluate the output of the code execution. You should determine if there were any bugs as well as report the empirical findings. Include essential information about the result, including warnings, errors, and the final metric.
Code <*Python code generated by the agent*>
Goals and Explanation <*explanation of the code*>
Execution Output <*terminal output*>

Response Template for Execution Result Summarization

is_bug true if the output log shows that the execution failed or has some bug, otherwise false
summary write a short summary (4-5 sentences) describing the empirical findings
output_abs select representative segments of the output log and mark the remainder as ellipses
metric If the code **ran successfully and produced submission.csv on full test set (i.e. not dummy or partial)**, report the value of the **final** validation metric. Otherwise, leave it null.
is_lower_better true if the metric should be minimized (i.e. a lower metric value is better, such as with MSE), false if the metric should be maximized (i.e. a higher metric value is better, such as with accuracy)

CoMind progressively optimizes the solution through trail and error. After the execution result is collected and summarized, it will try to revise the code by notifying the LLM:

Prompt for Consequent Code Revisions

Remaining steps: <*remaining_steps*>; Remaining time: <*remaining_time*> seconds
I ran your code and summarized the execution result: <*summary*>
Now, please choose your next action and propose code using the same response format as before. Remember, output a self-contained code, no part of it should be omitted. Keep the final validation metric same as the metric mentioned in the task description.

A) Fix runtime errors (if any)
B) Do hyperparameter tuning
C) Include ideas that were not implemented yet
D) Add possible improvements

E) Run on a larger scale (moderately increase training epochs, etc.). You should refer to the previous execution time we reported. Remember, your code will be killed if timeout.

B.4 Report Generation

Each sub-agent of CoMind compiles a comprehensive report and submits it to the report pool when time expires.

Prompt for Report Compilation

Please summarize the results and submit a comprehensive report.

Response Template for Report Compilation

pipeline A detailed description of the pipeline that generated the best results. All hyperparameters, training settings, model architectures, feature engineering, validation metric, and any other relevant information should be included. Describe potential improvements and future work.
summary A comprehensive evaluation of each individual component of the pipeline. For each component, summarize in the following format:
=== <*name of the component*> ===
Novelty: 0-10 (0: trivial, 10: clearly novel - major differences from existing well-known methods)
<*your rationale*>
Feasibility: 0-10 (0: almost impossible to implement and require extensive engineering, 10: Easy to implement)
<*your rationale*>
Effectiveness: 0-10 (0: minimal performance improvement, 10: very strong performance, significantly outperform most baselines)
<*your rationale*>
Efficiency: 0-10 (0: very slow, over-dependent on CPU and hard to produce meaningful results within the time limit, 10: high utilization of GPU)
<*your rationale*>
Confidence: 0-10 (0: no emprical results, not sure whether the evaluation is correct, 10: fully verified on large scale with abundant results)

C Case Study: Denoising Dirty Documents

C.1 Dataset Preparation

Besides the task description and datasets prepared in MLE-Bench, MLE-Live collects 59 public kernels and 19 discussions which are available on Kaggle and are posted before the competition ends.

C.1.1 Example of Public Kernel

```
"""
A simple feed-forward neural network that denoises one pixel at a time
"""
import numpy as np
import theano
import theano.tensor as T
import cv2
import os
import itertools

theano.config.floatX = 'float32'
```

```python

def load_image(path):
    return cv2.imread(path, cv2.IMREAD_GRAYSCALE)

def feature_matrix(img):
    """Converts a grayscale image to a feature matrix

    The output value has shape (<number of pixels>, <number of features>)
    """
    # select all the pixels in a square around the target pixel as features
    window = (5, 5)
    nbrs = [cv2.getRectSubPix(img, window, (y, x)).ravel()
            for x, y in itertools.product(range(img.shape[0]), range(img.shape[1]))]

    # add some more possibly relevant numbers as features
    median5 = cv2.medianBlur(img, 5).ravel()
    median25 = cv2.medianBlur(img, 25).ravel()
    grad = np.abs(cv2.Sobel(img, cv2.CV_16S, 1, 1, ksize=3).ravel())
    div = np.abs(cv2.Sobel(img, cv2.CV_16S, 2, 2, ksize=3).ravel())

... (omitted) ...

    # for fname in os.listdir('../input/test/'):
    for fname in ['1.png']:
        test_image = load_image(os.path.join('../input/test', fname))
        test_x = feature_matrix(test_image)

        y_pred, = predict(test_x)
        output = y_pred.reshape(test_image.shape)*255.0

        cv2.imwrite('original_' + fname, test_image)
        cv2.imwrite('cleaned_' + fname, output)


if __name__ == '__main__':
    main()
```

C.1.2 Example of Discussion

```
# Edge Diffraction in train_cleaned data
(Lance <TIER: N/A>) <p>I'm studying the pixels in train_cleaned data.  I attached a colorized blow-up
      version of part of the image train_cleaned/45.png.   The yellow pixels are any pixels that
      were not pure white ( != 0xFF gray scale) in image 45.png, the green was pure white (0xFF).</p>
   <p>So you see what looks like an edge diffraction line lining the outer edge of all the letters.</p>
   <p>Okay, maybe I got something wrong in my code.  Can anyone confirm this edge diffraction thing
        in the train_cleaned data, as for example the first word in train_cleaned/45.png (There). 
        You need to make the non-white (byte != 0xFF) pixels all a more contrasting color or you may not
        see it.</p>
   <p>I'm guessing that the clean png files were at some point scanned in using some kind of optical
        scanning machine which added these edge diffraction lines when the light diffracts off the edge
        of the black ink character.</p>
   ... (omitted) ...
   + (Rangel Dokov <TIER: MASTER>) <p>Yes, there is some noise, which doesn't look like it should be there
         in the clean set... I ran a test setting everything whiter that 0xF5 to 0xFF and the RMSE was
        0.005, which should be an upper bound on the effects from the halos. This will likely be large
        enough to make the top of the leaderboard a game of luck, but since this is just a playground
        competition I'm not terribly worried about it.</p>
```

C.2 Idea Selection

In our experiment settings, CoMind only accesses top-10 voted discussions and kernels and ignores the rest. Upon completion of this process, 7 ideas and 10 pipelines are generated. Below is an excerpt of the ideas and public pipelines.

```
(0) Use behaviour-based clustering of neural networks: cluster models by their error patterns and ensemble
      them for document enhancement
(1) Implement sliding-window patch-based models that take an input window and output multiple cleaned
     pixels simultaneously for both denoising and resolution enhancement
```

(2) Apply a Waifu2x-inspired deep convolutional neural network with gradually increasing filter counts (e.g., 1 -> 32 -> 64 -> 128 -> 256 -> 512 -> 1) and LeakyReLU activations for effective denoising
(3) Carefully initialize convolutional weights (e.g., stdv = sqrt(2/(kW*kH*nOutputPlane))) and use LeakyReLU to improve model convergence and performance
(4) Ensemble multiple models with different input preprocessing: combine outputs from a pure CNN, background-removed images, edge maps, and thresholded inputs to capture diverse noise characteristics
(5) Augment training data to simulate real-world 3D deformations and shadows on text, not just 2D noise, to better match test-time artifacts
(6) Account for systematic artifacts in 'clean' training data (e.g., single-pixel halos) by treating them as noise or adjusting targets accordingly during training

Public pipeline (0): - Pipeline: A simple feed-forward neural network that denoises one pixel at a time (Theano).
- Feature engineering: for each pixel extract a 5*5 window of gray values (neighbors), 5*5 median blur, 25*25 median blur, Sobel gradient and second-order derivative magnitudes, stack into a feature vector. Normalize features to [0,1].
- Model architecture: two-layer MLP; hidden layer size N_HIDDEN=10, tanh activation, output layer with custom activation clip(x+0.5,0,1).
- Training: MSE cost, stochastic gradient descent with learning rate 0.1, batch size 20, epochs 100. Validation on one image (3.png) at each epoch by RMSE.
- Prediction: apply same feature_matrix to test images, predict pixel values, reshape to full image, write out cleaned PNGs.

- Code abstract:
```
    def feature_matrix(img):
        window=(5,5)
        nbrs=[cv2.getRectSubPix(img,window,(y,x)).ravel()
              for x,y in itertools.product(range(img.shape[0]),range(img.shape[1]))]
        median5=cv2.medianBlur(img,5).ravel()
        median25=cv2.medianBlur(img,25).ravel()
        grad=np.abs(cv2.Sobel(img,cv2.CV_16S,1,1,ksize=3).ravel())
        div=np.abs(cv2.Sobel(img,cv2.CV_16S,2,2,ksize=3).ravel())
        misc=np.vstack((median5,median25,grad,div)).T
        features=np.hstack((nbrs,misc))
        return (features/255.).astype('float32')
    ...
    class Model(object):
        def __init__(...):
            self.layer1=Layer(...,n_in=...,n_out=N_HIDDEN,activation=T.tanh)
            self.layer2=Layer(...,n_in=N_HIDDEN,n_out=n_out,
                              activation=lambda x: T.clip(x+0.5,0,1))
        def cost(self,y): return T.mean((self.output-y)**2)
    ...
```
---------- PIPELINE SEPARATOR ----------

Public pipeline (1): - Pipeline: Matching image backgrounds in R (no ML model).
- Reads test PNGs in batches of 12 images.
- Flattens each into vectors of size 258*540, stacks as columns.
- For each pixel location, takes the maximum value across images as an estimate of background.
- Writes out background images as PNG.

- Code abstract:
```
    for(i in 1:4) {
      matches=seq(1,205,by=12)+(i-1)*3
      rawData=matrix(0,258*540,length(matches))
      for(j in seq_along(matches)){
        imgY=readPNG(file.path(testDir,paste0(matches[j],'.png')))
        rawData[,j]=as.vector(imgY[1:258,1:540])
      }
      background=matrix(apply(rawData,1,max),258,540)
      writePNG(background, paste0('background',matches[j],'.png'))
    }
    ...
```
---------- PIPELINE SEPARATOR ----------

Public pipeline (2): - Pipeline: Pixel-wise Random Forest regression (Python, chunk size=1e6).
- Feature engineering: pad image by mean value (padding=1); extract 3*3 neighborhood per pixel, flatten as features.
- Training data: load all train noisy images, compute features via joblib parallel (n_jobs=128), load targets as flattened clean pixel intensities/255.
- Model: sklearn.ensemble.RandomForestRegressor(warm_start=True, n_jobs=-1). Incrementally add one estimator at a time: split training rows into CHUNKSIZE=1e6 slices, in each slice increase n_estimators by 1 and fit on that slice.
- Prediction: extract test features similarly, generate index strings "image_row_col", predict pixel values, write submission CSV.

- Code abstract:
```
    def get_padded(img, padding=1):
        padval=int(round(img.mean()))
        ... return padded
    def get_features_for_image(img,padding=1):
        padded=get_padded(img,padding)
        return np.vstack([padded[i:i+3,j:j+3].reshape(1,-1)
                          for i in range(rows) for j in range(cols)])
    ...
    def get_model(X,y):
```

```
        model=RandomForestRegressor(n_estimators=0,warm_start=True,n_jobs=-1)
        for start,end in slices:
            model.set_params(n_estimators=model.n_estimators+1)
            model.fit(X[start:end],y[start:end])
        return model
    ...
---------- PIPELINE SEPARATOR ----------
... (omitted) ...
---------- PIPELINE SEPARATOR ----------
Public pipeline (9): - Pipeline: XGBoost with basic features in R.
  - Features: raw pixel value, kmeansThreshold binary, background_Removal feature.
  - Sample 25k pixels, build DMatrix, 5-fold CV, early stop, select rounds, train xgb.
  - Predict on test, clip, build id strings, write submission.csv.
- Code abstract:
    dat=cbind(y,raw=img2vec(imgX),thresholded=kmeansThreshold(imgX),
              backgroundRemoval=img2vec(background_Removal(imgX)))
    rows=sample(nrow(dat),25000)
    dtrain=xgb.DMatrix(dat[rows,-1],label=dat[rows,1])
    xgb.tab=xgb.cv(data=dtrain,nfold=5,early.stop.round=50)
    xgb.mod=xgboost(data=dtrain,nrounds=min.err.idx)
    yHat=predict(xgb.mod,newdata=as.matrix(x))
    ...
```

After generating ideas and public pipelines, CoMind initiates the first iteration. It begins brainstorming to propose more original strategies.

```
===SOLUTION_PATH_1===
Multi-Scale Wavelet-U-Net with Attention and Multi-Term Loss
Description: decompose each image via discrete wavelet transform (DWT) into low-/high-frequency bands,
      process each band in a dual-branch U-Net enhanced with channel- and spatial-attention, then invert
      the wavelet to reconstruct. Train end-to-end with a weighted sum of pixel-wise RMSE, wavelet-band
      consistency, and edge preservation losses.
- apply 2D DWT to noisy input -> obtain LL, LH, HL, HH sub-bands
- feed LL into a "coarse" encoder branch, feed concatenated LH/HL/HH into a "detail" encoder branch
- use a U-Net decoder to upsample each branch back to patch size, fuse via learned 1*1 convolutions
- insert Convolutional Block Attention Modules (CBAM) after each encoder and decoder block
- define loss = alpha*pixelRMSE(clean,output) + beta*bandRMSE(wavelet(clean),wavelet(output)) + gamma*
      edgeLoss(Sobel(clean),Sobel(output))
- train on full images with AdamW and a cosine-annealing LR schedule

===SOLUTION_PATH_2===
Stroke-Aware Conditional GAN with OCR-Guided Perceptual Loss
Description: build a conditional GAN (generator = deep residual encoder-decoder, discriminator = PatchGAN)
       that not only minimizes pixel loss but also preserves text strokes-enforce a stroke-level loss via a
       pre-trained small CNN classifier that predicts presence/width of strokes. Add an OCR-based
      perceptual loss: feed predictions through a frozen OCR engine embedding and minimize distance to
      clean embedding.
- implement generator as ResNet blocks + skip connections (64->128->256->128->64)
- implement discriminator as 70*70 PatchGAN to focus on local texture
- include L1 pixel loss + adversarial loss + stroke consistency loss (L1 between stroke-CNN features on
      clean vs. restored)
- freeze a small text-structure CNN (trained on binary masks) to extract stroke features
- run Tesseract (or lightweight OCR CNN) on restored vs. clean, extract penultimate-layer activations, add
       perceptual loss term
- train with R1 gradient penalty and spectral normalization on discriminator

===SOLUTION_PATH_3===
Joint Dictionary Learning + Non-Local Patch Aggregation
Description: learn paired dictionaries (D_noisy, D_clean) for small patches (e.g. 8*8) via coupled K-SVD.
      At test time, extract overlapping patches, compute sparse codes alpha under D_noisy via OMP,
      reconstruct clean patches = D_clean*alpha. Then, perform non-local means on the reconstructed patches
       to exploit self-similarity and average aggregates.
- sample a large bank of noisy/clean patch pairs, initialize D_noisy, D_clean with DCT basis
- run coupled K-SVD to minimize ||D_noisy*alpha - y_noisy|| + ||D_clean*alpha - y_clean|| w.r.t. D_noisy,
      D_clean,alpha
- at test time, for each image patch y_noisy, compute alpha via Orthogonal Matching Pursuit (sparsity <= k)

- reconstruct y_clean_est = D_clean*alpha for each patch
- perform block-matching to find K nearest patches per reference patch (Euclid dist), stack them
- aggregate reconstructed patches with non-local weights (e.g. Gaussian on reconstruction residual)

===SOLUTION_PATH_4===
Self-Supervised Blind Denoising via Noise2Void + Test-Time Adaptation
Description: exploit purely noisy data-train a small U-Net with masked pixel prediction (Noise2Void) on
      each test image at inference (test-time training). The network learns to predict a pixel from its
      context, gradually adapting to local noise statistics, then you run a forward pass to obtain the
      cleaned image. No clean target needed.
- define blind-spot or random masking scheme: mask 1% pixels per batch, replace with neighbors
- build a lightweight CNN (e.g. 5 down/up blocks with skip connections) that predicts a full image
- fine-tune this CNN on each test image for N_iter (e.g. 500 steps) using only masked L2 loss
```

```
- use data augmentation (rotations, flips) on the single test image to diversify contexts
- after adaptation, perform a clean forward pass without masking to get the denoised output
- optionally ensemble outputs from multiple random initializations to reduce variance
```

These strategies are then added to the idea pool. To remove similar ideas and decompose overlapped ideas, a reconstruction is performed subsequently. 9 ideas are preserved after the reconstruction.

```
(0) Use diverse ensembles by clustering models based on their error patterns and combining outputs from
     differently preprocessed inputs (e.g., raw, background-removed, edge maps, thresholded) to capture
     varied noise characteristics.
(1) Develop a sliding-window, patch-based model that takes an input region and predicts multiple denoised
     and super-resolved pixels simultaneously for efficient document enhancement.
(2) Implement a Waifu2x-inspired deep convolutional network with progressively increasing filter counts
     and LeakyReLU activations, initialized using He normalization for robust convergence in denoising
     tasks.
(3) Augment the training set with simulated 3D deformations, lighting variations, and shadows on text to
     better match real-world scanning artifacts.
(4) Identify and compensate for systematic artifacts in the "clean" training targets (e.g., single-pixel
     halos) by modeling them as noise or adjusting ground-truth labels during training.
(5) Build a wavelet-domain dual-branch U-Net: decompose inputs with 2D DWT into approximation and detail
     bands, encode separately, decode and fuse via learned 1*1 convolutions, integrate CBAM attention
     modules, and train end-to-end with a composite loss combining pixel RMSE, wavelet-band RMSE, and edge
      loss, optimized using AdamW with a cosine-annealing LR schedule.
(6) Apply a GAN-based restoration framework: use a ResNet-based generator and 70*70 PatchGAN discriminator,
      train with a combination of L1 pixel loss, adversarial loss, stroke-consistency loss (via a frozen
     stroke-feature CNN), and perceptual OCR-feature loss, incorporating R1 gradient penalty and spectral
     normalization.
(7) Leverage coupled dictionary learning and non-local aggregation: learn noisy and clean dictionaries
     simultaneously via coupled K-SVD, reconstruct patches using OMP, enhance self-similarity with block-
     matching, and fuse reconstructed patches with non-local weighting.
(8) Implement blind-spot test-time adaptation: randomly mask a small percentage of pixels per batch, train
      a lightweight CNN with skip connections on the single test image using masked L2 loss and
     augmentations, fine-tune per image for a few hundred steps, and optionally ensemble multiple
     initializations for variance reduction.
```

C.3 Idea Generation

Based on the idea pool, public pipelines and previous reports (currently unavailable), CoMind compiles 2 solution drafts.

~~~~
Pipeline 1 (Baseline CNN - U-Net with L1 loss and simple augmentations)
--------------------------------------------------------------------------------
Overview
- A lightweight U-Net trained on 256*256 patches of noisy/clean document pairs.
- Single-branch encoder-decoder with skip connections, ReLU activations, batch norm.
- Loss: L1 pixel-wise loss (less sensitive to outliers than MSE), optional total variation regularization.
- Optimizer: Adam; train on 1 A6000 within 1 hr.

Data Preparation
1. Read all noisy (input) and clean (target) train images, normalize intensities to [0,1].
2. Extract random 256*256 patches (stride = 128) with matching noisy/clean pairs.
3. Data augmentation: random horizontal/vertical flips, +-90 degree rotations.
4. Create PyTorch DataLoader with batch_size=16 (fits 48 GB) for ~100 k patches.

Network Architecture (PyTorch pseudocode)
```
class UNet(nn.Module):
    def __init__(self):
        super().__init__()
        # Encoder
        self.enc1 = DoubleConv(1, 64)
        self.enc2 = Down(64,128)
        self.enc3 = Down(128,256)
        self.enc4 = Down(256,512)
        # Bottleneck
        self.bottleneck = Down(512,512)
        # Decoder
        self.up4 = Up(1024,256)
        self.up3 = Up(512,128)
        self.up2 = Up(256,64)
        self.up1 = Up(128,64)
        self.final = nn.Conv2d(64,1,kernel_size=1)
    def forward(self,x):
        e1=self.enc1(x)
        e2=self.enc2(e1)
        e3=self.enc3(e2)
        e4=self.enc4(e3)
        b = self.bottleneck(e4)
~~~~

```
        d4=self.up4(b,e4)
        d3=self.up3(d4,e3)
        d2=self.up2(d3,e2)
        d1=self.up1(d2,e1)
        return torch.sigmoid(self.final(d1))
```
Helper modules:
- DoubleConv = (Conv2d -> BatchNorm2d -> ReLU) *2
- Down = (MaxPool2d -> DoubleConv)
- Up = (ConvTranspose2d for upsampling -> concatenate skip -> DoubleConv)

Training
- Loss = L1Loss(output, target) + lambda*TV(output) (lambda=1e-5 for smoothness).
- Optimizer = Adam(lr=1e-3, weight_decay=1e-5).
- LR schedule: ReduceLROnPlateau(monitor=val_loss, factor=0.5, patience=5).
- Train for up to 50 epochs; early-stop if val_loss stagnates.
- Validation: hold out 10% patches to monitor RMSE.

Inference
- For each test image (e.g., 540*258), slide 256*256 window with stride=128, predict, and average overlapping outputs.
- Threshold nothing; output raw [0,1] floats per pixel.

Compute Budget
- ~100 k patches, batch 16, ~6 k steps per epoch. On A6000: ~2-3 min/epoch => 50 epochs ~ 2 hr; with early stopping < 1 hr.

Pipeline 2 (Advanced Wavelet U-Net with CBAM and Composite Loss)

Overview
- Dual-branch U-Net operating in wavelet domain (Haar DWT) to explicitly denoise tonal and textural components.
- CBAM (Convolutional Block Attention Modules) to adaptively weigh spatial/channel features.
- Loss = alpha*L1_pixel + beta*L2_wavelet + gamma*EdgeLoss.
- Optimizer = AdamW + CosineAnnealingLR.

Data Preparation
- Same as Pipeline 1 (256*256 patches + augmentations).
- On-the-fly DWT: for each noisy patch, compute one-level Haar DWT -> yields approximation (A) and details (H,V,D).

Network Architecture
(implemented in PyTorch)
```
class WaveletUNet(nn.Module):
    def __init__(self):
        super().__init__()
        # Shared CBAM-Res blocks for Approx and Detail branches
        self.encA1 = CBAMResBlock(1,64)
        self.encD1 = CBAMResBlock(3,64)
        self.pool = nn.MaxPool2d(2)
        self.encA2 = CBAMResBlock(64,128)
        self.encD2 = CBAMResBlock(64,128)
        # Bottleneck
        self.bottleneck = CBAMResBlock(256,256)
        # Decoder
        self.up2 = UpRes(256,128)
        self.up1 = UpRes(128,64)
        self.final = nn.Conv2d(64,4,kernel_size=1)
    def forward(self, x):
        # x shape: [B,1,H,W]
        A, (H,V,D) = haar_dwt(x) # PyWavelets or custom conv
        d0 = torch.cat([self.encA1(A), self.encD1(torch.cat([H,V,D],dim=1))],dim=1)
        p1 = self.pool(d0)
        d1 = torch.cat([self.encA2(p1[:, :128]), self.encD2(p1[:, 128:])],dim=1)
        b = self.bottleneck(self.pool(d1))
        u2 = self.up2(b,d1)
        u1 = self.up1(u2,d0)
        out = self.final(u1) # [B,4,H,W]
        # Reconstruct denoised x via inverse DWT
        A_hat, H_hat,V_hat,D_hat = out[:,0:1], out[:,1:2], out[:,2:3], out[:,3:4]
        return ihaar_dwt((A_hat, (H_hat,V_hat,D_hat)))
```
Key building blocks:
- CBAMResBlock: residual block with Conv-BN-ReLU->Conv-BN, plus CBAM (channel & spatial attention after second Conv).
- UpRes: ConvTranspose2d upsample->concatenate skip->CBAMResBlock.

Loss
- L1_pixel = L1(output, target).
- L2_wavelet = L2(DWT(output) - DWT(target)).

```
- EdgeLoss = L1(Sobel(output), Sobel(target)).
- TotalLoss = alpha*L1_pixel + beta*L2_wavelet + gamma*EdgeLoss (alpha=1, beta=0.5, gamma=0.2).

Training
- Optimizer: AdamW(lr=1e-3, weight_decay=1e-4).
- Scheduler: CosineAnnealingLR(T_max=50).
- Batch size=8 (higher memory due to dual branches).
- Epochs=50, early-stop by composite loss on val set.

Inference
- Same sliding-window approach as Pipeline 1.

Compute Budget
- CBAM and wavelet layers add ~1.5* compute but within A6000 48 GB: patch batch=8 yields ~250 steps/epoch,
      so 50 epochs ~2 hr. Use early-stop or T_max=30 to finish within 1 hr.


References
- CBAM: Woo et al., "CBAM: Convolutional Block Attention Module" (ECCV 2018).
- Haar DWT: PyWavelets or custom conv filters.
- Edge loss: Sobel via fixed conv kernels.

Expected Benefits
- Explicit noise suppression in high-frequency bands.
- Attention modules focus on text strokes and edge restoration.
- Composite loss balances global structure and fine details.
```

C.4 Implementation and Improvement

In our experiment, 2 parallel instances are invoked to implement the solution drafts respectively. Each instance has a quota of 20 steps, 3 hours in total and 1 hour per execution. After each execution, another summarization LLM is instructed to analyze the terminal output.

C.4.1 Example of Terminal Output

```
Loaded 115 train images
Sampled train/val patch coords: 80000 10000
.../python3.11/site-packages/torch/optim/lr_scheduler.py:62: UserWarning: The verbose parameter is
      deprecated. Please use get_last_lr() to access the learning rate.
  warnings.warn(
Traceback (most recent call last):
  File ".../agent_0.py", line 231, in <module>
    for xb, yb in train_loader:
  File ".../python3.11/site-packages/torch/utils/data/dataloader.py", line 708, in __next__
    data = self._next_data()
           ^^^^^^^^^^^^^^^^^
  File ".../python3.11/site-packages/torch/utils/data/dataloader.py", line 1480, in _next_data
    return self._process_data(data)
           ^^^^^^^^^^^^^^^^^^^^^^^^
  File ".../python3.11/site-packages/torch/utils/data/dataloader.py", line 1505, in _process_data
    data.reraise()
  File ".../python3.11/site-packages/torch/_utils.py", line 733, in reraise
    raise exception
ValueError: Caught ValueError in DataLoader worker process 0.
Original Traceback (most recent call last):
  File ".../python3.11/site-packages/torch/utils/data/_utils/worker.py", line 349, in _worker_loop
    data = fetcher.fetch(index) # type: ignore[possibly-undefined]
           ^^^^^^^^^^^^^^^^^^^^
  File ".../python3.11/site-packages/torch/utils/data/_utils/fetch.py", line 52, in fetch
    data = [self.dataset[idx] for idx in possibly_batched_index]
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File ".../python3.11/site-packages/torch/utils/data/_utils/fetch.py", line 52, in <listcomp>
    data = [self.dataset[idx] for idx in possibly_batched_index]
            ~~~~~~~~~~~~^^^^^
  File ".../agent_0.py", line 121, in __getitem__
    n_patch = torch.from_numpy(n_patch).unsqueeze(0)
              ^^^^^^^^^^^^^^^^^^^^^^^^^
ValueError: At least one stride in the given numpy array is negative, and tensors with negative strides
      are not currently supported. (You can probably work around this by making a copy of your array with
      array.copy().)
```

C.4.2 Example of Output Summary

```
Terminal output (truncated):
Loaded 115 train images
```

```
Sampled train/val patch coords: 80000 10000
.../python3.11/site-packages/torch/optim/lr_scheduler.py:62: UserWarning: ...
Traceback (most recent call last):
  File ".../agent_0.py", line 231, in <module>
    for xb, yb in train_loader:
  ...
ValueError: At least one stride in the given numpy array is negative, and tensors with negative strides
      are not currently supported. (You can probably work around this by making a copy of your array with
      array.copy().)
Execution summary:
The training script fails during the first epoch's data loading. After loading 115 train images and
      sampling 80k train and 10k validation patches, the DataLoader worker crashes with a ValueError due to
       negative strides when converting flipped NumPy patches to tensors. As a result, the pipeline halts
      before any training or validation metrics are produced and no submission.csv is generated.
Execution time: 3.81 seconds.
```

C.4.3 Example of Generated Snippet

The best code snippet generated in the first iteration of CoMind is:

```python
import os, glob, random, math
from PIL import Image
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import Dataset, DataLoader
from torchvision import transforms


# -- 1) Haar DWT / inverse DWT --
class HaarDWT(nn.Module):
    def __init__(self):
        super().__init__()
        wt = torch.zeros(4, 1, 2, 2)
        wt[0, 0, :, :] = 0.5
        wt[1, 0, 0, 0], wt[1, 0, 0, 1], wt[1, 0, 1, 0], wt[1, 0, 1, 1] = (
            0.5,
            -0.5,
            0.5,
            -0.5,
        )
        wt[2, 0, 0, 0], wt[2, 0, 0, 1], wt[2, 0, 1, 0], wt[2, 0, 1, 1] = (
            0.5,
            0.5,
            -0.5,
            -0.5,
        )
        wt[3, 0, 0, 0], wt[3, 0, 0, 1], wt[3, 0, 1, 0], wt[3, 0, 1, 1] = (
            0.5,
            -0.5,
            -0.5,
            0.5,
        )
        self.register_buffer("weight", wt)

    def forward(self, x):
        return F.conv2d(x, self.weight, stride=2)


class HaarIDWT(nn.Module):
    def __init__(self):
        super().__init__()
        wt = HaarDWT().weight.clone()
        self.conv = nn.ConvTranspose2d(4, 1, 2, stride=2, bias=False)
        self.conv.weight.data.copy_(wt)
        self.conv.weight.requires_grad_(False)

```

```
    def forward(self, coeffs):
        return self.conv(coeffs)


# -- 2) Sobel edge for EdgeLoss --
class Sobel(nn.Module):
    def __init__(self):
        super().__init__()
        kx = torch.tensor(
            [[1, 0, -1], [2, 0, -2], [1, 0, -1]], dtype=torch.float32
        ).view(1, 1, 3, 3)
        ky = torch.tensor(
            [[1, 2, 1], [0, 0, 0], [-1, -2, -1]], dtype=torch.float32
        ).view(1, 1, 3, 3)
        self.register_buffer("wx", kx)
        self.register_buffer("wy", ky)

    def forward(self, x):
        gx = F.conv2d(x, self.wx, padding=1)
        gy = F.conv2d(x, self.wy, padding=1)
        return torch.sqrt(gx * gx + gy * gy + 1e-6)


# -- 3) CBAM, ResBlock, UpRes, WaveletUNet --
class CBAM(nn.Module):
    def __init__(self, c, r=16, k=7):
        super().__init__()
        self.mlp = nn.Sequential(
            nn.Linear(c, c // r, bias=False),
            nn.ReLU(inplace=True),
            nn.Linear(c // r, c, bias=False),
        )
        self.spatial = nn.Conv2d(2, 1, kernel_size=k, padding=k // 2, bias=False)

    def forward(self, x):
        b, c, h, w = x.shape
        avg = F.adaptive_avg_pool2d(x, 1).view(b, c)
        mx = F.adaptive_max_pool2d(x, 1).view(b, c)
        ca = torch.sigmoid(self.mlp(avg) + self.mlp(mx)).view(b, c, 1, 1)
        x2 = x * ca
        avgc = x2.mean(1, True)
        maxc, _ = x2.max(1, True)
        sa = torch.sigmoid(self.spatial(torch.cat([avgc, maxc], 1)))
        return x2 * sa


class CBAMResBlock(nn.Module):
    def __init__(self, inp, outp):
        super().__init__()
        self.conv1 = nn.Conv2d(inp, outp, 3, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(outp)
        self.conv2 = nn.Conv2d(outp, outp, 3, padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(outp)
        self.cbam = CBAM(outp)
        self.skip = (
            nn.Conv2d(inp, outp, 1, bias=False) if inp != outp else nn.Identity()
        )
        self.relu = nn.ReLU(inplace=True)

    def forward(self, x):
        idt = self.skip(x)
        out = self.relu(self.bn1(self.conv1(x)))
        out = self.bn2(self.conv2(out))
        out = self.cbam(out) + idt
        return self.relu(out)
```

```python


class UpRes(nn.Module):
    def __init__(self, ic, sc, oc):
        super().__init__()
        self.up = nn.ConvTranspose2d(ic, oc, 2, stride=2)
        self.block = CBAMResBlock(oc + sc, oc)

    def forward(self, x, skip):
        x = self.up(x)
        if x.shape[-2:] != skip.shape[-2:]:
            x = F.interpolate(
                x, size=skip.shape[-2:], mode="bilinear", align_corners=False
            )
        return self.block(torch.cat([x, skip], 1))


class WaveletUNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.dwt = HaarDWT()
        self.idwt = HaarIDWT()
        self.eA1 = CBAMResBlock(1, 64)
        self.eD1 = CBAMResBlock(3, 64)
        self.pool = nn.MaxPool2d(2)
        self.eA2 = CBAMResBlock(64, 128)
        self.eD2 = CBAMResBlock(64, 128)
        self.b = CBAMResBlock(256, 256)
        self.u2 = UpRes(256, 256, 128)
        self.u1 = UpRes(128, 128, 64)
        self.final = nn.Conv2d(64, 4, 1)

    def forward(self, x):
        A, H, V, D = self.dwt(x).chunk(4, 1)
        a1 = self.eA1(A)
        d1 = self.eD1(torch.cat([H, V, D], 1))
        d0 = torch.cat([a1, d1], 1)
        p1 = self.pool(d0)
        pA, pD = p1[:, :64], p1[:, 64:]
        a2 = self.eA2(pA)
        d2 = self.eD2(pD)
        d1b = torch.cat([a2, d2], 1)
        b = self.b(self.pool(d1b))
        u2 = self.u2(b, d1b)
        u1 = self.u1(u2, d0)
        out = self.final(u1)
        return self.idwt(out)


# -- 4) OCRDataset for full training --
class OCRDataset(Dataset):
    def __init__(self, noisy, clean, pp, ps, mode):
        self.noisy = noisy
        self.clean = clean
        self.pp = pp
        self.ps = ps
        self.mode = mode
        self.toT = transforms.ToTensor()

    def __len__(self):
        return len(self.noisy) * self.pp

    def __getitem__(self, idx):
        ii = idx // self.pp
        n = Image.open(self.noisy[ii]).convert("L")
```

```python
        c = Image.open(self.clean[ii]).convert("L")
        tn, tc = self.toT(n), self.toT(c)
        H, W = tn.shape[-2], tn.shape[-1]
        y = random.randint(0, H - self.ps)
        x = random.randint(0, W - self.ps)
        tn = tn[:, y : y + self.ps, x : x + self.ps]
        tc = tc[:, y : y + self.ps, x : x + self.ps]
        if self.mode == "train":
            if random.random() < 0.5:
                tn, tc = torch.flip(tn, [-1]), torch.flip(tc, [-1])
            if random.random() < 0.5:
                tn, tc = torch.flip(tn, [-2]), torch.flip(tc, [-2])
            k = random.randint(0, 3)
            if k > 0:
                tn, tc = torch.rot90(tn, k, [1, 2]), torch.rot90(tc, k, [1, 2])
        return tn, tc


# -- 5) Prepare train/val split --
all_no = sorted(glob.glob("./input/train/*.png"))
all_cl = sorted(glob.glob("./input/train_cleaned/*.png"))
idxs = list(range(len(all_no)))
random.seed(42)
random.shuffle(idxs)
split = int(0.9 * len(idxs))
tr_idxs, va_idxs = idxs[:split], idxs[split:]
noisy_tr = [all_no[i] for i in tr_idxs]
clean_tr = [all_cl[i] for i in tr_idxs]
noisy_va = [all_no[i] for i in va_idxs]
clean_va = [all_cl[i] for i in va_idxs]

# -- 6) DataLoaders --
train_ds = OCRDataset(noisy_tr, clean_tr, pp=25, ps=256, mode="train")
val_ds = OCRDataset(noisy_va, clean_va, pp=1, ps=256, mode="val")
train_loader = DataLoader(
    train_ds, batch_size=8, shuffle=True, num_workers=4, pin_memory=True
)
val_loader = DataLoader(
    val_ds, batch_size=1, shuffle=False, num_workers=2, pin_memory=True
)

# -- 7) Model, optimizer, scheduler, loss w/ tuned weights --
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = WaveletUNet().to(device)
dwt = HaarDWT().to(device)
sobel = Sobel().to(device)
opt = torch.optim.AdamW(model.parameters(), lr=1e-3, weight_decay=1e-4)
sched = torch.optim.lr_scheduler.CosineAnnealingLR(opt, T_max=30)

# Tuned weights
alpha = 0.769634
beta = 0.430968
gamma = 0.250750


def composite_loss(o, t):
    l1 = F.l1_loss(o, t)
    l2 = F.mse_loss(dwt(o), dwt(t))
    e1 = F.l1_loss(sobel(o), sobel(t))
    return alpha * l1 + beta * l2 + gamma * e1


# -- 8) Train + validate --
best_rmse = 1e9
patience = 5
```

```
wait = 0
for epoch in range(1, 31):
    model.train()
    tloss = 0.0
    for xb, yb in train_loader:
        xb, yb = xb.to(device), yb.to(device)
        pred = model(xb)
        loss = composite_loss(pred, yb)
        opt.zero_grad()
        loss.backward()
        opt.step()
        tloss += loss.item()
    sched.step()

    # sliding-window validation
    model.eval()
    rmse_sum = 0.0
    stride, ps = 128, 256
    with torch.no_grad():
        for nf, cf in zip(noisy_va, clean_va):
            imn = Image.open(nf).convert("L")
            imc = Image.open(cf).convert("L")
            tn = transforms.ToTensor()(imn).unsqueeze(0).to(device)
            tc = transforms.ToTensor()(imc).unsqueeze(0).to(device)
            _, _, H, W = tn.shape
            acc = torch.zeros_like(tn)
            cnt = torch.zeros_like(tn)
            xs = list(range(0, W - ps + 1, stride)) + [W - ps]
            ys = list(range(0, H - ps + 1, stride)) + [H - ps]
            for x in xs:
                for y in ys:
                    p = tn[:, :, y : y + ps, x : x + ps]
                    out = model(p)
                    acc[:, :, y : y + ps, x : x + ps] += out
                    cnt[:, :, y : y + ps, x : x + ps] += 1
            rec = acc / cnt
            mse = F.mse_loss(rec, tc).item()
            rmse_sum += math.sqrt(mse)
    val_rmse = rmse_sum / len(noisy_va)
    print(
        f"Epoch {epoch}: train_loss={tloss/len(train_loader):.4f}, val_RMSE={
            val_rmse:.6f}"
    )

    if val_rmse < best_rmse:
        best_rmse = val_rmse
        wait = 0
        torch.save(model.state_dict(), "./working1/best.pth")
    else:
        wait += 1
        if wait >= patience:
            print("Early stopping.")
            break

print("Best validation RMSE:", best_rmse)

# -- 9) Inference & submission --
model.load_state_dict(torch.load("./working1/best.pth"))
model.eval()
test_files = sorted(glob.glob("./input/test/*.png"))
with open("./working1/submission.csv", "w") as fout:
    fout.write("id,value\n")
    for tf in test_files:
        im = Image.open(tf).convert("L")
        t = transforms.ToTensor()(im).unsqueeze(0).to(device)
```

```python
        _, _, H, W = t.shape
        acc = torch.zeros_like(t)
        cnt = torch.zeros_like(t)
        xs = list(range(0, W - ps + 1, stride)) + [W - ps]
        ys = list(range(0, H - ps + 1, stride)) + [H - ps]
        with torch.no_grad():
            for x in xs:
                for y in ys:
                    p = t[:, :, y : y + ps, x : x + ps]
                    out = model(p)
                    acc[:, :, y : y + ps, x : x + ps] += out
                    cnt[:, :, y : y + ps, x : x + ps] += 1
        rec = (acc / cnt).clamp(0, 1).cpu().numpy()[0, 0]
        img_id = os.path.splitext(os.path.basename(tf))[0]
        for r in range(H):
            for c in range(W):
                fout.write(f"{img_id}_{r+1}_{c+1},{rec[r,c]:.6f}\n")

print("Submission written to ./working1/submission.csv")
```

C.5 Report Generation

The report generated by the second instance of CoMind is:

```
Pipeline:
Our final pipeline is an Advanced Wavelet U-Net with CBAM attention modules, Composite Loss, and sliding-
      window inference. Specifically:

1. Data Preparation
   - Read all 112 noisy/clean train pairs, split 90/10 train/val by file indices (random seed 42).
   - On-the-fly extract 25 random 256*256 patches per train image each epoch, with random horizontal/
       vertical flips and 90 degree rotations.
   - Validation uses a single full-image sliding-window pass (256 patches with 50% overlap) on the 10% hold
       -out.

2. Model Architecture
   - Haar DWT and inverse DWT implemented as fixed convolutional filters (LL, LH, HL, HH bands).
   - Dual-branch encoder: separate CBAMResBlocks for the approximation (1 channel) and detail (3 channels)
       coefficients.
   - Two-level downsampling with MaxPool, interleaved CBAMResBlocks, and a 256-channel bottleneck.
   - Decoder with ConvTranspose upsampling and CBAMResBlock skip-connections (UpRes), recombining wavelet
       bands.
   - Final 1*1 convolution to 4 coefficient channels and inverse DWT back to image domain.

3. Composite Loss
   - L1(pixels) weighted by alpha=0.769634
   - L2(wavelet coefficients) weighted by beta=0.430968
   - L1(edge Sobel) weighted by gamma=0.250750

4. Training
   - Optimizer: AdamW(lr=1e-3, weight_decay=1e-4).
   - Scheduler: CosineAnnealingLR(T_max=30).
   - Batch size 8, 30 epochs, early stopping (patience=5) based on full-image RMSE on the val set.

5. Inference
   - Sliding-window full-image inference with 256*256 patches, stride=128, averaging overlapped pixels.
   - Generate submission.csv with id=image_row_col and float values.

6. Hyperparameter tuning
   - Optuna over (alpha, beta, gamma) on a 20-train/5-val mini-split for 10 trials, each with 2 epochs and
       10 patches/image.
   - Best found: alpha=0.769634, beta=0.430968, gamma=0.250750.

Final validation RMSE on 10% hold-out: ~0.012685.

References: CBAM (Woo et al., ECCV 2018), Haar DWT.

Summary:
- Haar DWT:
  Novelty: 2
  Rationale: Uses standard 1-level Haar wavelet decomposition.
  Feasibility: 10
  Rationale: Straightforward fixed conv implementation in PyTorch.
  Effectiveness: 6
```

```
  Rationale: Explicit frequency separation aids denoising of high-frequency noise.
  Efficiency: 9
  Rationale: Lightweight fixed kernels on GPU.
  Confidence: 8
  Rationale: Verified on toy and full runs.

- CBAMResBlock:
  Novelty: 5
  Rationale: Incorporates both channel & spatial attention for context-aware features.
  Feasibility: 8
  Rationale: Moderate complexity but supported by existing PyTorch modules.
  Effectiveness: 7
  Rationale: Improves focus on text strokes, empirically lowers RMSE.
  Efficiency: 7
  Rationale: Adds overhead (~1.5*), but fits within GPU budget.
  Confidence: 9
  Rationale: Confirmed across toy and full training.

- UpRes (ConvTranspose + CBAM):
  Novelty: 3
  Rationale: Standard upsampling with skip-connections and attention.
  Feasibility: 9
  Rationale: Simple extension of ResBlock.
  Effectiveness: 6
  Rationale: Maintains detail in reconstruction.
  Efficiency: 8
  Rationale: Reasonable GPU use.
  Confidence: 8
  Rationale: Working in end-to-end script.

- Wavelet UNet overall:
  Novelty: 7
  Rationale: Dual-branch U-Net in wavelet domain combining DWT, CBAM, skip connections.
  Feasibility: 7
  Rationale: Moderate engineering but fully implemented.
  Effectiveness: 8
  Rationale: Achieved RMSE ~0.0127, strong denoising.
  Efficiency: 7
  Rationale: 25 patches*8 batch; 30 epochs in ~20min.
  Confidence: 8
  Rationale: Reproducible on A6000.

- Composite Loss:
  Novelty: 6
  Rationale: Balances pixel, frequency, and edge domains.
  Feasibility: 8
  Rationale: Simple weighted sum.
  Effectiveness: 8
  Rationale: Lower RMSE than L1 or L2 alone.
  Efficiency: 7
  Rationale: Small overhead for extra transforms.
  Confidence: 9
  Rationale: Verified in tuning and full runs.

- Data Augmentation:
  Novelty: 3
  Rationale: Flips and rotations.
  Feasibility: 10
  Rationale: Trivial to implement.
  Effectiveness: 5
  Rationale: Standard but helpful for invariance.
  Efficiency: 9
  Rationale: CPU-level only.
  Confidence: 9
  Rationale: Common practice.

- CosineAnnealingLR & EarlyStopping:
  Novelty: 4
  Rationale: Standard learning-rate decay and val-based stop.
  Feasibility: 10
  Rationale: Built-in PyTorch.
  Effectiveness: 6
  Rationale: Helps convergence and prevents overfit.
  Efficiency: 9
  Rationale: Minimal overhead.
  Confidence: 9
  Rationale: Observed smoothed training curves.

- Sliding-window Inference:
  Novelty: 3
  Rationale: Ensures full-image coverage.
```

```
  Feasibility: 10
  Rationale: Straightforward loops.
  Effectiveness: 7
  Rationale: Accurate RMSE measurement, smooth outputs.
  Efficiency: 8
  Rationale: Reasonable inference time (~10 min).
  Confidence: 9
  Rationale: Verified end-to-end.

- Hyperparameter Tuning (Optuna):
  Novelty: 4
  Rationale: Automated search of loss weights.
  Feasibility: 8
  Rationale: Low-cost mini-training.
  Effectiveness: 7
  Rationale: Found better weights vs. defaults.
  Efficiency: 5
  Rationale: 10 trials * 2 epochs ~8 min.
  Confidence: 7
  Rationale: Verified chosen parameters.

Weaknesses and Suggestions:
1. EMA & TTA Bug: The EMA + 4-way TTA implementation caused RMSE blow-ups, likely due to incorrect
     accumulation. Before reintroducing, validate TTA on toy splits or isolate EMA without TTA.
2. Multi-Level Wavelets: Extend to 2-3 levels of DWT to capture coarser scales, with separate branches per
      level.
3. Advanced Loss Terms: Incorporate perceptual (VGG) or structural (SSIM) losses to encourage visual
     fidelity.
4. Dynamic Loss Weights: Schedule (alpha,beta,gamma) during training (e.g., start with high L1, gradually
     emphasize wavelet).
5. Mixed Precision & Gradient Checkpointing: Further reduce memory and speed up with AMP and checkpointing
      deep layers.
6. Deeper/Ubiquitous Attention: Explore transformer-style attention (Swin, ViT) in encoder for global
     context.
7. Ensemble Models: Train multiple random seeds or architectures (ResUNet, UNet++) and average outputs.
8. Multi-Scale Inference: Use patches at multiple scales or whole-image passes via down/upscaling for
     context.
9. Adaptive Overlap: Use weights at edges of patches to smoothly blend overlaps (Gaussian windows).
10. Distributed Hyperparameter Search: Scale Optuna across more trials or use BOHB for architecture search.

Implementing these will likely push RMSE even lower and improve robustness across document noise patterns.
Full code: ...(omitted)...
```

C.6 The Second Iteration

Upon receiving two reports generated in the previous stage, CoMind proceeds to its second iteration. Following the brainstorming and idea refinement phases, the current idea pool is presented below. Notably, entries (3), (4), (5), and (7) are newly proposed by the agent, while the remaining ideas have been refined from those in the previous iteration.

```
(0) Patch-based convolutional super-resolution denoiser: a sliding-window, patch-based model that predicts
      multiple denoised and super-resolved pixels per patch. Architecture inspired by Waifu2x with
     progressively increasing filter counts, LeakyReLU activations, and He initialization for robust
     convergence.
(1) Advanced wavelet-domain dual-branch U-Net with CBAM attention and composite loss: decompose inputs via
      fixed 1-level DWT (LL, LH, HL, HH bands), encode approximation and detail separately with CBAM
     ResBlocks, decode and fuse via 1*1 convolutions, and train end-to-end using a weighted sum of pixel
     L1, wavelet-band L2, and edge L1 losses. Optimized with AdamW and cosine-annealing LR scheduling.
(2) GAN-based restoration framework: a ResNet-based generator and 70*70 PatchGAN discriminator trained
     with combined losses-L1 pixel loss, adversarial loss, stroke-consistency loss (via frozen stroke-
     feature CNN), and perceptual OCR-feature loss. Includes R1 gradient penalty and spectral
     normalization for stability.
(3) Masked autoencoder with vision transformer for denoising: patchify each image into non-overlapping
     square tokens, randomly mask a high percentage, pretrain a ViT encoder (12 layers, hidden 768, 12
     heads) plus light transformer decoder on L2 reconstruction of dirty images, then append an MLP head
     and fine-tune end-to-end on noisy->clean pairs with L1 pixel + differentiable OCR-confidence loss.
     Employ random block dropout and color jitter during fine-tuning; at inference use full-image encoding
      or averaged mask schedules.
(4) Conditional diffusion-based restoration: define a forward Gaussian-noise diffusion schedule, train a 5-
     level U-Net conditioned on the dirty image via channel concatenation and FiLM/cross-attention of
     sinusoidal timestep embeddings. Use the standard DDPM MSE loss with classifier-free guidance, and
     sample with a deterministic DDIM sampler (~50 steps). Optionally post-process with bilateral or
     median filtering to remove speckles.
(5) Learnable spectral gating in the Fourier domain: compute the 2D FFT of the dirty image, split its
     spectrum into low/mid/high radial bands, apply learnable complex masks per band, and modulate each by
      gate scalars predicted by a lightweight CNN on the dirty image. Recombine via inverse FFT and train
     end-to-end with L2 pixel loss plus a spectral-smoothness regularizer on the masks.
```

(6) Hypernetwork-modulated U-Net: extract per-image noise statistics (mean, std, skew, kurtosis, histogram bins), feed into an MLP hypernetwork that outputs FiLM scale (gamma) and shift (beta) parameters for selected convolutional feature maps of a base U-shaped CNN. Randomly augment noise levels during training; train end-to-end on noisy->clean with L1 loss and a small regularizer pushing gamma->1, beta->0. At inference compute stats per image, generate FiLM params, and denoise via the modulated U-Net.
(7) Blind-spot test-time adaptation: for each test image, randomly mask a subset of pixels and fine-tune a lightweight CNN with skip connections on the single image using masked L2 loss and augmentations for a few hundred gradient steps. Optionally ensemble multiple random initializations to reduce variance.

(8) Multi-model ensemble with diverse preprocessing: cluster trained models by their error patterns and combine their outputs. Apply different preprocessing pipelines (raw, background-removed, edge maps, thresholded) to the input, denoise with clustered sub-ensembles, and fuse predictions for robustness across noise characteristics.
(9) Enhanced augmentation and target refinement: simulate realistic scanning artifacts by applying 3D text deformations, lighting variations, and shadows to clean images. Identify and compensate for systematic artifacts in the provided 'clean' targets (e.g., single-pixel halos) by either modeling them as noise or adjusting ground-truth labels during training.

And solution drafts generated in this iteration are:

```
Pipeline 1: ResNet-34 Encoder U-Net with Multi-Scale Edge & Total-Variation Loss

Overview:
A robust baseline using a pretrained ResNet-34 backbone as a U-Net encoder fused with a light-weight
      decoder. Combines L1 loss, Sobel edge loss at multiple scales, and a total-variation regularizer to
      preserve text strokes while smoothing background noise. Mixed precision training and sliding-window
      inference ensure the entire pipeline runs in ~45 min on an A6000.

1. Data Preparation
 - Read all train noisy/clean PNGs, normalize to [0,1].
 - Extract on-the-fly 256*256 patches: random crop + random horizontal/vertical flips + 90 degree rotations
      .
 - 90/10 split by file indices (seed=42). Use batch size 8-16.

2. Model Architecture
 - Encoder: torchvision.models.resnet34(pretrained=True), first conv modified to 1->64 channels.
 - Decoder: four upsampling stages (ConvTranspose2d + Conv2d+BN+ReLU) mirroring ResNet blocks, with skip-
      connections from encoder layers.
 - Final conv 64->1 + Sigmoid.

3. Loss Function
 Let y_hat and y be predictions and targets.
 - L1Loss(y_hat,y)
 - Edge loss: L1 between Sobel(y_hat) and Sobel(y) at both full resolution and half resolution (downsample
      by 2).
 - TV: lambda*TV(y_hat) where TV = mean(|\nabla xy_hat|+|\nabla yy_hat|).
 Total loss = alpha*L1 + beta*Edge_full + gamma*Edge_half + delta*TV, e.g. alpha=1.0, beta=0.5, gamma=0.25,
       delta=1e-5.

4. Optimization
 - Optimizer: AdamW(lr=1e-3, weight_decay=1e-4).
 - Scheduler: CosineAnnealingLR(T_max=25).
 - Mixed precision via torch.cuda.amp.
 - Early stopping on validation RMSE (patience=5).

5. Inference & Submission
 - Perform sliding-window inference on each test image with 256*256 patches, stride=128.
 - Average overlapping patches.
 - Clamp outputs to [0,1], write submission.csv with id=image_row_col.

Compute budget: ~20 min train + ~5 min inference.

Pipeline 2: Laplacian-Pyramid Multi-Scale Residual U-Net with Pyramid Loss

Overview:
A novel pyramid-domain network that decomposes images into multi-scale Laplacian bands, denoises each band
      via shared-weight residual blocks, and merges them back. Multi-level L1 losses focus the model on
      both coarse structures and fine text details. Efficient and fully end-to-end in PyTorch, training
      finishes in ~50 min on an A6000.

1. Data & Augmentation
 - Same data split and patch sampling (256*256, flips, rotations).

2. Laplacian Pyramid Transform (LPT)
 - On the fly, for each patch: create 2-level Gaussian pyramid using avg-pooling (scale 1->0.5->0.25), then
       compute Laplacian bands L0=(I-upsample(I/2)), L1=(I/2-upsample(I/4)), and a residual low band L2=I
      /4.
 - Stack [L0,L1,L2] as 3 input channels.
```

```
3. Network
 - Encoder-decoder U-Net with 3-channel input and 3-channel output, 4 down/upsampling stages,
     64->128->256->512->256->128->64 channel widths.
 - Each block: two 3*3 Conv+BN+ReLU and a residual skip connection.

4. Reconstruction
 - Output is 3 residual corrections R0,R1,R2 to each band.
 - Reconstruct clean patch:
   a0 = L0 + R0; a1 = L1 + R1; a2 = L2 + R2;
   I = a0 + up(a1) + up(up(a2)).

5. Loss Function
 - Per-band L1 losses: w0*L1(a0,H0) + w1*L1(a1,H1) + w2*L1(a2,H2), where Hk are ground-truth bands and w0
     =1.0, w1=0.5, w2=0.25.
 - Pixel L1: lambda_pix*L1(I, Igt) with lambda_pix=0.1.

6. Training
 - AdamW(lr=5e-4, weight_decay=1e-5), one-cycle scheduler (max_lr=5e-4, epochs=30).
 - Batch size 8, mixed precision.
 - Early stopping on val RMSE (patience=6).

7. Inference
 - On full test images, run sliding-window with 256*256 patches, stride=128.
 - For each patch, apply pyramid -> model -> reconstruct -> accumulate and average.
 - Clamp to [0,1], write submission.csv.

Compute budget: ~25 min training + ~8 min inference.
```

The best code proposed in the second iteration is:

```python
import os, random, csv
import numpy as np
from glob import glob
from PIL import Image

import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import Dataset, DataLoader
from torchvision import transforms, models as tv_models
from torch.cuda.amp import autocast, GradScaler

# 1) Reproducibility
SEED = 42
random.seed(SEED)
np.random.seed(SEED)
torch.manual_seed(SEED)
torch.cuda.manual_seed_all(SEED)

DEVICE = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# 2) Paths
TRAIN_NOISY = "./input/train"
TRAIN_CLEAN = "./input/train_cleaned"
TEST_DIR = "./input/test"
WORK_DIR = "./working0"
os.makedirs(WORK_DIR, exist_ok=True)

# 3) Hyperparameters
PATCH_SIZE = 256
STRIDE = 128
BATCH_SIZE = 12
LR, WD = 1e-3, 1e-4
MAX_EPOCHS = 50
PATIENCE = 10
T_MAX = 50 # for LR scheduler

# 4) Loss-weight constants (from tuning)
w1, w2, w3, w4 = 1.0, 0.5, 0.25, 1e-5
lambda_aux = 0.4394633936788146
```

```
lambda_mse = 0.1312037280884873
lambda_ssim = 0.031198904067240532
lambda_ssim2 = lambda_ssim / 2


# 5) Dataset + augmentations
class OCRDataset(Dataset):
    def __init__(self, noisy_list, clean_list, ps, train):
        self.noisy, self.clean = noisy_list, clean_list
        self.ps, self.train = ps, train
        self.to_tensor = transforms.ToTensor()
        self.aug = transforms.Compose(
            [
                transforms.RandomChoice(
                    [
                        transforms.RandomHorizontalFlip(1.0),
                        transforms.RandomVerticalFlip(1.0),
                        transforms.RandomRotation(90),
                        transforms.RandomRotation(180),
                        transforms.RandomRotation(270),
                    ]
                ),
                transforms.RandomApply([transforms.GaussianBlur(3, (0.1, 2.0))], p
                    =0.3),
                transforms.RandomApply([transforms.RandomAdjustSharpness(2.0)], p
                    =0.3),
            ]
        )

    def __len__(self):
        return len(self.noisy)

    def __getitem__(self, i):
        n = Image.open(self.noisy[i]).convert("L")
        c = Image.open(self.clean[i]).convert("L")
        w, h = n.size
        # pad
        if w < self.ps or h < self.ps:
            pad = (0, 0, max(0, self.ps - w), max(0, self.ps - h))
            n = transforms.functional.pad(n, pad, fill=255)
            c = transforms.functional.pad(c, pad, fill=255)
            w, h = n.size
        # crop
        if self.train:
            x = random.randint(0, w - self.ps)
            y = random.randint(0, h - self.ps)
        else:
            x = (w - self.ps) // 2
            y = (h - self.ps) // 2
        n = n.crop((x, y, x + self.ps, y + self.ps))
        c = c.crop((x, y, x + self.ps, y + self.ps))
        if self.train and random.random() < 0.5:
            n = self.aug(n)
            c = self.aug(c)
        return self.to_tensor(n), self.to_tensor(c)


# 6) Prepare train/val split
noisy_files = sorted(glob(f"{TRAIN_NOISY}/*.png"))
clean_files = [f"{TRAIN_CLEAN}/" + os.path.basename(x) for x in noisy_files]
N = len(noisy_files)
idx = list(range(N))
random.shuffle(idx)
ntr = int(0.9 * N)
tr_idx, va_idx = idx[:ntr], idx[ntr:]
```

```python
train_noisy = [noisy_files[i] for i in tr_idx]
train_clean = [clean_files[i] for i in tr_idx]
val_noisy = [noisy_files[i] for i in va_idx]
val_clean = [clean_files[i] for i in va_idx]

train_ds = OCRDataset(train_noisy, train_clean, PATCH_SIZE, train=True)
val_ds = OCRDataset(val_noisy, val_clean, PATCH_SIZE, train=False)
train_loader = DataLoader(
    train_ds, batch_size=BATCH_SIZE, shuffle=True, num_workers=4, pin_memory=True
)
val_loader = DataLoader(
    val_ds, batch_size=BATCH_SIZE, shuffle=False, num_workers=4, pin_memory=True
)

# 7) Sobel, TV, SSIM helpers
sob_x = (
    torch.tensor([[1, 0, -1], [2, 0, -2], [1, 0, -1]], dtype=torch.float32)
    .view(1, 1, 3, 3)
    .to(DEVICE)
)
sob_y = sob_x.transpose(2, 3)


def sobel(x):
    gx = F.conv2d(x, sob_x, padding=1)
    gy = F.conv2d(x, sob_y, padding=1)
    return torch.sqrt(gx * gx + gy * gy + 1e-6)


def total_variation(x):
    dh = (x[:, :, 1:, :] - x[:, :, :-1, :]).abs().mean()
    dw = (x[:, :, :, 1:] - x[:, :, :, :-1]).abs().mean()
    return dh + dw


def ssim_map(a, b, C1=0.01**2, C2=0.03**2):
    mu_a = F.avg_pool2d(a, 3, 1, 1)
    mu_b = F.avg_pool2d(b, 3, 1, 1)
    sa = F.avg_pool2d(a * a, 3, 1, 1) - mu_a * mu_a
    sb = F.avg_pool2d(b * b, 3, 1, 1) - mu_b * mu_b
    sab = F.avg_pool2d(a * b, 3, 1, 1) - mu_a * mu_b
    num = (2 * mu_a * mu_b + C1) * (2 * sab + C2)
    den = (mu_a * mu_a + mu_b * mu_b + C1) * (sa + sb + C2)
    return num / (den + 1e-8)


def ssim_loss(a, b):
    return 1.0 - ssim_map(a, b).mean()


# 8) loss_terms
l1_loss = nn.L1Loss()
mse_loss = nn.MSELoss()


def loss_terms(pred, target):
    L1v = l1_loss(pred, target)
    MSEv = mse_loss(pred, target)
    Ef = l1_loss(sobel(pred), sobel(target))
    p2, t2 = F.avg_pool2d(pred, 2), F.avg_pool2d(target, 2)
    Eh = l1_loss(sobel(p2), sobel(t2))
    TVv = total_variation(pred)
    return L1v, MSEv, Ef, Eh, TVv
```

```python
# 9) Model w/ deep supervision
class ResUNetDS(nn.Module):
    def __init__(self):
        super().__init__()
        r34 = tv_models.resnet34(pretrained=True)
        self.enc0 = nn.Conv2d(1, 64, 7, 2, 3, bias=False)
        self.enc0.weight.data = r34.conv1.weight.data.mean(dim=1, keepdim=True)
        self.bn0, self.relu0, self.pool0 = r34.bn1, r34.relu, r34.maxpool
        self.enc1, self.enc2 = r34.layer1, r34.layer2
        self.enc3, self.enc4 = r34.layer3, r34.layer4

        def up(i, o):
            return nn.ConvTranspose2d(i, o, 2, 2)

        def cb(i, o):
            return nn.Sequential(
                nn.Conv2d(i, o, 3, 1, 1, bias=False),
                nn.BatchNorm2d(o),
                nn.ReLU(inplace=True),
                nn.Conv2d(o, o, 3, 1, 1, bias=False),
                nn.BatchNorm2d(o),
                nn.ReLU(inplace=True),
            )

        self.up4, self.dec4 = up(512, 256), cb(256 + 256, 256)
        self.up3, self.dec3 = up(256, 128), cb(128 + 128, 128)
        self.up2, self.dec2 = up(128, 64), cb(64 + 64, 64)
        self.aux_up, self.aux_out = up(64, 64), nn.Conv2d(64, 1, 1)
        self.up1, self.dec1 = up(64, 64), cb(64 + 64, 64)
        self.up0, self.outc = up(64, 64), nn.Conv2d(64, 1, 1)
        self.sig = nn.Sigmoid()

    def forward(self, x):
        x0 = self.relu0(self.bn0(self.enc0(x)))
        x1 = self.pool0(x0)
        x2 = self.enc1(x1)
        x3 = self.enc2(x2)
        x4 = self.enc3(x3)
        x5 = self.enc4(x4)

        d4 = self.dec4(torch.cat([self.up4(x5), x4], dim=1))
        d3 = self.dec3(torch.cat([self.up3(d4), x3], dim=1))
        d2 = self.dec2(torch.cat([self.up2(d3), x2], dim=1))
        aux = self.sig(self.aux_out(self.aux_up(d2)))
        d1 = self.dec1(torch.cat([self.up1(d2), x0], dim=1))
        main = self.sig(self.outc(self.up0(d1)))
        return main, aux


model = ResUNetDS().to(DEVICE)

# 10) Optimizer, scheduler, scaler
optimizer = torch.optim.AdamW(model.parameters(), lr=LR, weight_decay=WD)
scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=T_MAX)
scaler = GradScaler()

# 11) Training + snapshot saving
best_rmse = float("inf")
patience = 0
snap_epochs = set([10, 20, 30, 40, 50])

for epoch in range(1, MAX_EPOCHS + 1):
    model.train()
    train_loss = 0.0
    for noisy_img, clean_img in train_loader:
```

```
        noisy_img, clean_img = noisy_img.to(DEVICE), clean_img.to(DEVICE)
        optimizer.zero_grad()
        with autocast():
            main_pred, aux_pred = model(noisy_img)
            L1v, MSEv, Ef, Eh, TVv = loss_terms(main_pred, clean_img)
            s1 = ssim_loss(main_pred, clean_img)
            p2, t2 = F.avg_pool2d(main_pred, 2), F.avg_pool2d(clean_img, 2)
            s2 = ssim_loss(p2, t2)
            main_loss = (
                w1 * L1v
                + lambda_mse * MSEv
                + w2 * Ef
                + w3 * Eh
                + w4 * TVv
                + lambda_ssim * s1
                + lambda_ssim2 * s2
            )
            aux_up = F.interpolate(
                aux_pred,
                size=clean_img.shape[-2:],
                mode="bilinear",
                align_corners=False,
            )
            La, Ma, Ea, Eh2, TVa = loss_terms(aux_up, clean_img)
            sa = ssim_loss(aux_up, clean_img)
            pa, ca = F.avg_pool2d(aux_up, 2), F.avg_pool2d(clean_img, 2)
            sa2 = ssim_loss(pa, ca)
            aux_loss = (
                w1 * La
                + lambda_mse * Ma
                + w2 * Ea
                + w3 * Eh2
                + w4 * TVa
                + lambda_ssim * sa
                + lambda_ssim2 * sa2
            )
            loss = main_loss + lambda_aux * aux_loss
        scaler.scale(loss).backward()
        scaler.step(optimizer)
        scaler.update()
        train_loss += loss.item()
    scheduler.step()

    # validation
    model.eval()
    se, count = 0.0, 0
    with torch.no_grad():
        for noisy_img, clean_img in val_loader:
            noisy_img, clean_img = noisy_img.to(DEVICE), clean_img.to(DEVICE)
            with autocast():
                pred, _ = model(noisy_img)
            se += ((pred - clean_img) ** 2).sum().item()
            count += pred.numel()
    val_rmse = np.sqrt(se / count)
    print(
        f"Epoch {epoch}: TrainLoss={train_loss/len(train_loader):.4f}, ValRMSE={val_rmse:.6f}"
    )

    # best + snapshot
    if val_rmse < best_rmse:
        best_rmse = val_rmse
        torch.save(model.state_dict(), os.path.join(WORK_DIR, "best_full.pth"))
        patience = 0
    else:
```

```
            patience += 1
    if epoch in snap_epochs:
        torch.save(model.state_dict(), os.path.join(WORK_DIR, f"snap_{epoch}.pth"))
    if patience >= PATIENCE:
        print("Early stopping.")
        break

print("Best validation RMSE:", best_rmse)

# 12) Ensemble load
ckpts = ["best_full.pth"] + sorted(
    [f for f in os.listdir(WORK_DIR) if f.startswith("snap_")],
    key=lambda x: int(x.split("_")[1].split(".")[0]),
)[-2:]
ensemble_nets = []
for ck in ckpts:
    net = ResUNetDS().to(DEVICE)
    net.load_state_dict(torch.load(os.path.join(WORK_DIR, ck)))
    net.eval()
    ensemble_nets.append(net)


# 13) Sliding-window ensemble inference
def ensemble_infer(img_arr):
    h, w = img_arr.shape
    inp = torch.from_numpy(img_arr / 255.0).unsqueeze(0).unsqueeze(0).to(DEVICE)
    ph = (PATCH_SIZE - h % STRIDE) % STRIDE
    pw = (PATCH_SIZE - w % STRIDE) % STRIDE
    inp = F.pad(inp, (0, pw, 0, ph), mode="reflect")
    _, _, H, W = inp.shape
    out = torch.zeros_like(inp)
    wt = torch.zeros_like(inp)
    for y in range(0, H - PATCH_SIZE + 1, STRIDE):
        for x in range(0, W - PATCH_SIZE + 1, STRIDE):
            patch = inp[:, :, y : y + PATCH_SIZE, x : x + PATCH_SIZE]
            preds = []
            with torch.no_grad(), autocast():
                for net in ensemble_nets:
                    p, _ = net(patch)
                    preds.append(p)
            avg_p = torch.stack(preds, 0).mean(0)
            out[:, :, y : y + PATCH_SIZE, x : x + PATCH_SIZE] += avg_p
            wt[:, :, y : y + PATCH_SIZE, x : x + PATCH_SIZE] += 1.0
    out = out / wt
    out = out[:, :, :h, :w]
    return out.detach().cpu().numpy().squeeze()


# 14) Write submission.csv
submission_path = os.path.join(WORK_DIR, "submission.csv")
with open(submission_path, "w", newline="") as f:
    writer = csv.writer(f)
    writer.writerow(["id", "value"])
    for tf in sorted(
        glob(f"{TEST_DIR}/*.png"), key=lambda x: int(os.path.basename(x).split(".")
            [0])
    ):
        img_id = os.path.basename(tf).split(".")[0]
        img = np.array(Image.open(tf).convert("L"), dtype=np.float32)
        den = ensemble_infer(img)
        H, W = den.shape
        for i in range(H):
            for j in range(W):
                writer.writerow([f"{img_id}_{i+1}_{j+1}", f"{den[i,j]:.6f}"])
print("Submission saved to", submission_path)
```

1962