

EEMA: Experience-Evolving Meta Agents for AutoML

Kai Tian^{1,2,*}, Weizhi Wang^{1,2,*}, Kaikai Zhao², Che Jiang^{1,2}, Junlin Yang^{1,2}
Zhenzhao Yuan^{1,2}, Jincheng Zhong^{1,2}, Guoli Jia², Yuru Wang^{1,2}, Xinwei Long²
Ning Ding², Bowen Zhou², Kaiyan Zhang^{1,†}

¹Horizon Research, Frontis.AI ²Tsinghua University

*Equal contribution †Corresponding author

zhangkaiyan@frontis.cn

Abstract

Open machine learning engineering tasks require agents to interpret data, build models, debug code, and produce valid competition submissions. Although current coding agents can generate runnable programs, they often rediscover solutions across tasks and repeat engineering mistakes across feedback rounds. We propose EEMA, an experience-evolving meta-agent framework that organizes accumulated experience by purpose and routes it during multi-round solving. Its static component distills public competition material and historical failure traces into compact Skills: task-level modeling priors for improving solution quality and failure-prevention contracts for maintaining execution validity. Its dynamic component, Experience Routed Search (ERS), operates above a lower-level coding agent. At each round, ERS selects a draft, debug, or improve action, routes the corresponding Skill, and updates task memory, allowing cross-task experience to guide code generation while task-level feedback shapes later decisions. On MLE-bench Lite (22 tasks), EEMA achieves a 77.3% medal rate, 20.1% average leaderboard Grade, and 96.0% execution success, outperforming all baselines including GPT-5.5-xhigh. Ablations show that task Skills improve solution quality, failure-prevention Skills improve reliability, and ERS coordinates both effectively. The materials will be released at <https://github.com/FrontisAI/OpenRSI>.

1 Introduction

Automated machine learning is shifting from closed pipeline tuning to open-ended machine learning engineering (MLE). Classic systems search over fixed spaces of algorithms, hyperparameters, or pipeline components (Thornton et al., 2013; Feurer et al., 2015; Olson et al., 2016). Open MLE tasks instead require agents to interpret natural-language instructions, inspect heterogeneous data, select models, write end-to-end

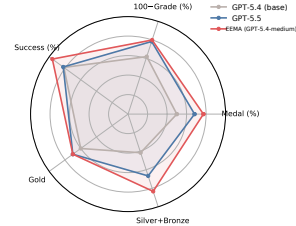


Figure 1: Results on MLE-bench Lite. Higher is better on all axes; *Grade* is plotted as $100 - \text{Grade}$.

training and inference code, and revise that code after errors or scores arrive. There is no fixed interface: a wrong column name, submission order, or metric can invalidate an otherwise reasonable solution. MLE-bench and recent MLE environments make this setting measurable (Chan et al., 2025; Qiang et al., 2025a). A central question is how prior engineering experience can be organized and reused to support open AutoML agents.

The challenge extends beyond code generation. Recent coding and AutoML agents already use execution feedback, search, and memory (Jiang et al., 2025; Nam et al., 2025; Liu et al., 2025). Yet many strong choices come from cross-task experience. Tabular tasks often require careful sample-ID handling and validation splits. Image tasks often require augmentation and resolution handling. Text tasks require tradeoffs between fast baselines and heavier models. Human competitors carry these patterns as meta-knowledge and update their strategies after failures. An agent confined to the current context may rediscover problems already solved elsewhere. At the same time, simply injecting more outside knowledge is insufficient: in multi-round search, the agent must decide when to fix errors, rebuild a stable baseline, refine near the current best, or switch method families.

We study a direct question: how can an AutoML agent both exploit accumulated experience and let that experience evolve within the current task? We decompose this into two sub-problems. The first

concerns the origin and form of experience: where it comes from and how to encode it so that an agent can act on it. The second concerns selection and adaptation: how experience is routed, compressed, and updated during multi-round solving. The first sub-problem is cross-task meta-learning; the second is in-task experience evolution.

We propose EEMA, an experience-evolving meta-agent framework for machine learning engineering. The static component distills reusable Skills from public competition material and past failure traces. A Skill is a compact action-oriented guide: it states how to model a class of tasks and which engineering errors should be avoided early. The dynamic component is Experience Routed Search, or ERS. ERS sits above a lower-level coding agent. It chooses an action type, routes a Skill, writes a plan contract, and updates task memory. The planning stage compresses the Skill, task memory, and search intent into an auditable contract, while the coding stage writes a solution program from this contract. Thus cross-task experience guides individual code-generation rounds, and task feedback reshapes how experience is used later. Figure 2 provides an overview of the offline experience construction and runtime evolution process, while Figure 1 summarizes the resulting aggregate gains.

Our contributions are as follows:

- We build a reusable experience base by distilling expert competition material and failure traces into compact Skills.
- We propose EEMA, with ERS as its dynamic search component, to route experience and update task memory across solving rounds.
- We evaluate EEMA on MLE-bench Lite, showing that routed experience raises medal coverage from 50.0% to 77.3% and improves average leaderboard grade from 38.0% to 20.1%, while maintaining 96.0% execution success.

2 Related Work

Automated machine learning and MLE benchmarks. Classic AutoML systems frame automation as algorithm selection, hyperparameter optimization, or pipeline search: Auto-WEKA jointly optimizes algorithms and hyperparameters, auto-sklearn adds meta-learning and ensembling, and

TPOT searches pipelines with genetic programming (Thornton et al., 2013; Feurer et al., 2015; Olson et al., 2016). OpenML established shared evaluation infrastructure for machine learning and AutoML (Vanschoren et al., 2014), while recent machine learning engineering benchmarks turn open-ended data science work into LLM agent tasks. MLE-bench evaluates agents on Kaggle-style competitions that require valid submissions (Chan et al., 2025); MLE-Dojo and MLE-Smith extend this direction with interactive environments and automated task construction (Qiang et al., 2025a,b). AIRS-Bench further broadens evaluation from single code generation to practical ML workflows and research-science loops (Lupidi et al., 2026). NatureBench further extends this setting to cross-disciplinary scientific tasks distilled from Nature-family publications, evaluating whether coding agents can reproduce or improve upon published methods in standardized environments (Wang et al., 2026b). We do not introduce a new benchmark. Instead, we study how transferable experience can be organized and routed while solving these open AutoML tasks.

Search and feedback in coding agents. Recent AutoML agents increasingly treat problem solving as iterative search in code space. AIDE maintains a tree of candidate scripts and alternates among drafting, debugging, and improvement (Jiang et al., 2025); MLE-STAR combines search with code-block ablation and targeted refinement (Nam et al., 2025); and AIRA-dojo compares search policies on MLE-bench (Toledo et al., 2025). Other systems improve the search process itself: reinforcement learning has been used to train AutoML agents from task feedback (Yang et al., 2025a), PACEvolve++ studies test-time learning for evolutionary search agents (Yan et al., 2026), MARTI-MARS² combines multi-agent self-search with reinforcement learning for code generation (Wang et al., 2026a), and Frontis-MA1 further aligns training and inference around learned program-evolution operators, composing Draft, Improve, Debug, and Crossover into long-horizon MLE search (Yang et al., 2026). ERS shares the use of execution feedback and multi-round search, but it controls a different unit. Rather than only selecting the next edit or candidate program, it decides which kind of experience should enter the next plan and records how each branch differs from the current best program.

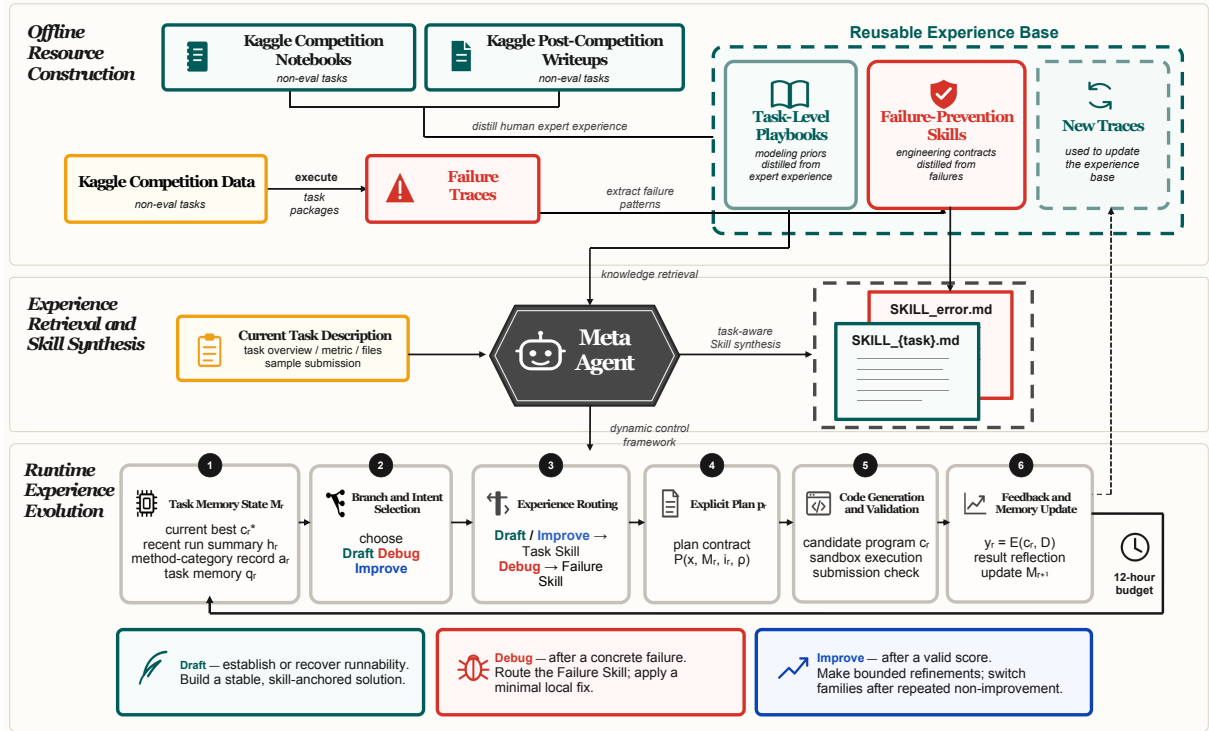


Figure 2: **Overview of EEMA.** Offline experience is distilled into task-level and failure-prevention Skills. At runtime, ERS iteratively selects draft, debug, or improve, routes the corresponding Skill, writes an explicit plan, executes a candidate program, receives feedback, and updates task memory.

Experience reuse, memory, and managed skills.

Our work also connects to retrieval and memory for LLM agents (Jiang et al., 2026). Retrieval-augmented generation injects external knowledge into generation (Lewis et al., 2020), while Reflexion stores verbal feedback for later actions (Shinn et al., 2023). In MLE settings, ML-Master, AutoMLGen, and KompeteAI explore multi-trajectory reasoning, fine-grained optimization, and reuse of past or external experience (Liu et al., 2025; Du et al., 2025; Kulibaba et al., 2025). Broader autonomous data science systems similarly emphasize managed agent capabilities: R&D-Agent links research ideas, implementation, and empirical feedback (Yang et al., 2025b); Agentic-imodels evolves interpretable modeling tools (Singh et al., 2026); and SkillsVote studies the collection, recommendation, and evolution of agent skills (Liu et al., 2026). We follow this shift from static prompts to managed capabilities, but specialize it for open AutoML runs by distilling short Skills and routing them according to search state and task memory.

3 Cross-Task Experience

Cross-task experience specifies what an agent should know before a new task starts. We encode it as Skills: short action-oriented texts with applicability conditions, key checks, and executable advice. From a meta-learning view, a Skill is a task prior extracted from past tasks.

3.1 Sources and Leakage Control

We use public machine learning competition material as the source of experience. The main source is Kaggle. This material includes data files, metrics, submission formats, and community solutions. It covers tabular, image, text, audio, time-series, and multimodal tasks. To reduce evaluation leakage, the static experience base excludes competition material that matches MLE-bench evaluation tasks. This includes data, discussions, notebooks, solutions, leaderboards and writeups.

The current expert base has two types of material. The notebook side covers 413 Kaggle competitions and 2,076 documents. The writeup side covers 520 Kaggle competitions and 2,714 documents.

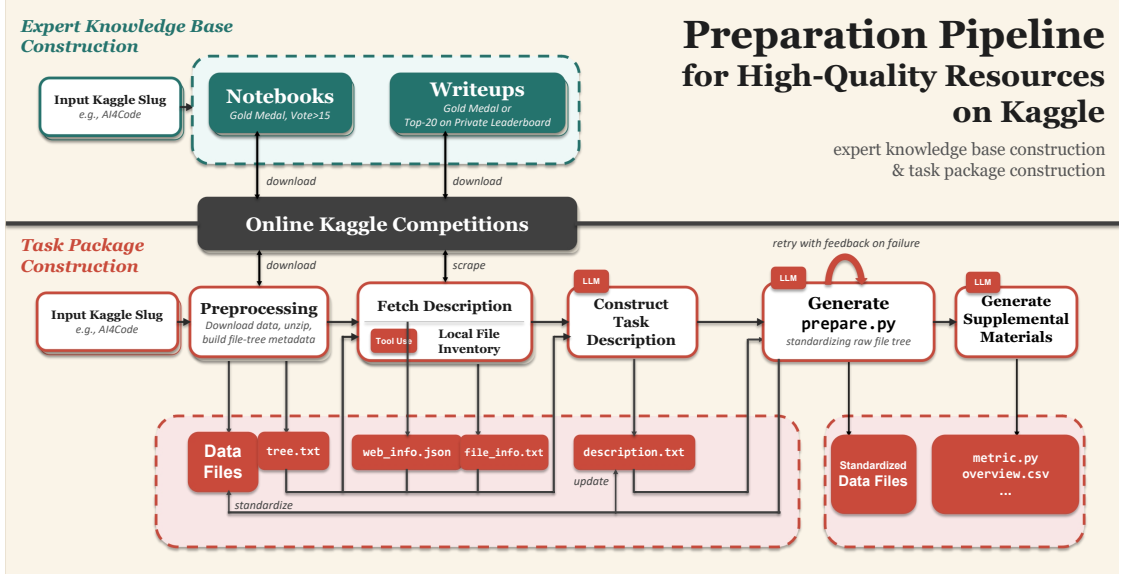


Figure 3: Preparation pipeline for high-quality Kaggle resources. The upper branch collects notebooks and writeups for the expert knowledge base; the lower branch constructs standardized task packages with data files, descriptions, file metadata, preparation scripts, metrics, and supplemental materials.

Source	Competitions	Documents
Notebook	413	2076
Writeup	520	2714
All	593(deduped)	4790

Table 1: Size of the Kaggle expert experience base.

3.2 Task-Level Expert Skills

Task-level Skills come from successful experience. We first group notebooks and writeups by data modality and task type. We then distill category playbooks. A category playbook summarizes transferable choices for preprocessing, feature engineering, models, training, inference, and post-processing. Given a target task, the system selects content that matches the task description, metric, data structure, and resource limits. The result is a task-level Skill. This is a transferable initialization learned from a task family, rather than search from scratch.

Let $\mathcal{K}_{m,t}$ be the expert experience set for modality m and task type t . A task-level Skill is

$$S_{\text{task}} = g_{\theta}(\mathcal{K}_{m,t}, d),$$

where d is the visible target-task description. It includes the instructions, file structure, sample submission, and a public training-data summary. The function g_{θ} is the distillation process. The output Skill should meet three conditions. It should be relevant to the current task. It should guide code,

such as how to preserve row identity, choose validation splits, or handle probability outputs. It should also be concise enough to leave context space for the task description, logs, and code.

3.3 Skill Form and Selection

A Skill is not a long retrieved document. It is a compact contract for action with four parts: scope, checks, recommended actions, and avoid rules. Scope states when the Skill applies; checks list facts to verify from files or logs; recommended actions give modeling or engineering steps; avoid rules name common traps. This turns many expert notes into a small set of choices that a coding agent can follow during planning.

Selection is conservative. A target task may match several modality and task labels. We select the main task-level Skill by the target metric and data type, then keep only advice supported by visible files or the task statement. Advice that requires hidden leaderboard knowledge is removed, keeping the Skill useful without turning it into a loose summary of public solutions.

Length is another constraint. In an open AutoML run, the context must also contain task instructions, data summaries, recent logs, the plan, and code. We therefore prefer short rules over detailed recipes: a Skill may require preserving an ID column or checking class imbalance, but should not copy a full notebook or force a fixed loss before the label distribution is known.

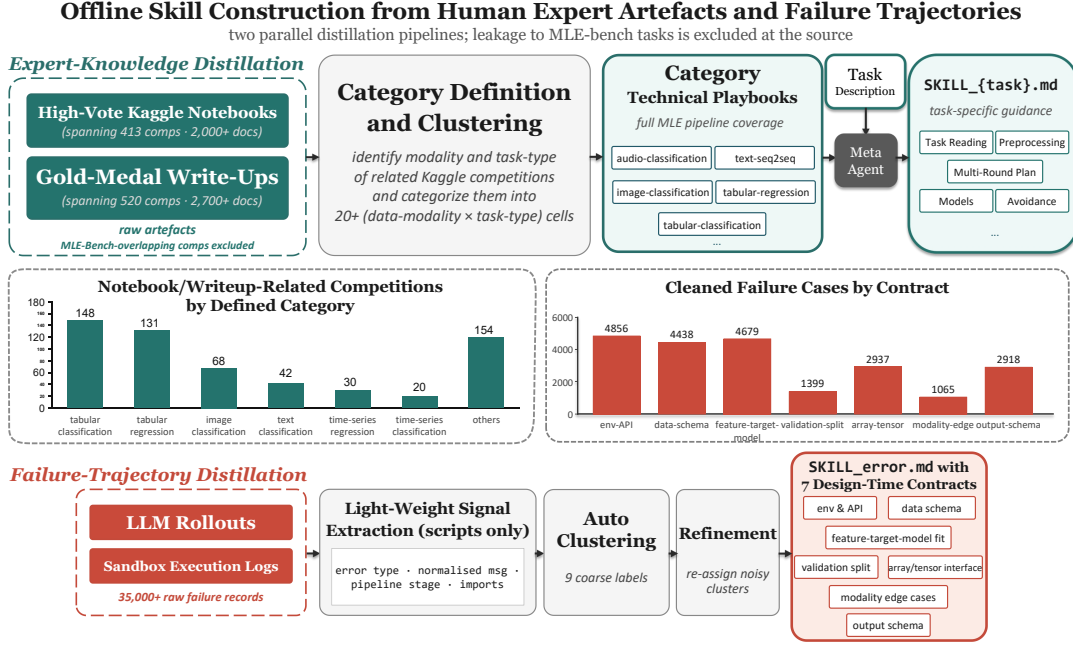


Figure 4: Skill construction pipeline. The system clusters expert experience and builds task-level Skills. Failure-prevention Skills are distilled from failure traces collected during large-scale task runs.

3.4 Failure-Prevention Skills

Successful experience raises the ceiling, but many failures happen before scoring. To reduce such failures, we built our own Kaggle competition data normalization pipeline and used MLE-Smith as a supplementary task-construction source (Qiang et al., 2025b). After large-scale generation and filtering, 2156 task packages are selected. We then ran agents on these non-evaluation tasks and distilled failure-prevention Skills from 35,615 raw failure traces.

The automatic clustering stage analyzes failed steps and finds common error types. They include environment APIs, data schemas, feature and model compatibility, array and tensor interfaces, output schemas, validation splits, and modality boundaries. We compress them into seven contracts: environment and API, data schema, feature-target and model compatibility, validation split, array-tensor and batch interface, modality boundary, and output schema.

A failure-prevention Skill checks basic engineering facts before code is written. Column names and file structure should be inferred from actual loading results. Submission row order and ID mapping should be preserved from the start. Library versions and optional arguments should be checked in the current environment. The Skill

turns past failures into new search constraints. Failure experience also becomes meta-knowledge.

Failure Skills are deliberately different from task Skills. A task Skill tries to improve the solution quality. A failure Skill tries to keep the run valid. This split matters because the two goals can conflict. A stronger model may need extra packages, longer training, or more complex input pipelines. These choices can increase the chance of failure. ERS uses this split later. It routes task Skills to draft and improve actions, and failure Skills to debug actions.

4 In-Task Experience Evolution

Cross-task Skills provide strong initial experience. ERS lets experience keep changing during the current task. Feedback in machine learning engineering is mixed. It may be a submission format error, a training crash, or a score plateau. ERS maps this feedback to three actions: draft, debug, and improve. Draft is anchored by the task-specific Skill and aims to implement a high-ROI stable recipe while satisfying data, resource, and submission contracts. Debug focuses on the latest failure. Improve focuses on score gains after a current best solution exists.

4.1 Problem Definition

Given a task x , public data D , evaluation environment E , and total budget B , the agent generates program c_r at round r . It receives feedback

$$y_r = E(c_r, D).$$

The feedback includes run status, validation score, error logs, and submission checks. Multi-round solving can be viewed as search with memory. The state at round r is

$$M_r = (c_r^*, h_r, a_r, q_r),$$

where c_r^* is the current best program, h_r is a recent run summary, a_r is a record of recent method categories, and q_r is task-level memory. ERS chooses an action type b_r , a search intent i_r , and a Skill source s_r from state M_r . It then writes a plan contract p_r , from which the lower-level agent generates a candidate program c_r . Thus, ERS adapts how experience is used, while the lower-level agent handles code generation.

The action type is in

$$b_r \in \{D, R, I\}.$$

Here D means draft. It builds or restores a strong runnable solution from the task-specific Skill, with schema, resource, and submission safeguards. R means debug. It handles the latest failure. I means improve. It raises the score of an existing runnable solution. The search intent gives a more specific action inside the same type. Examples include rebuilding a stable baseline, fixing an error, making a local improvement to the current best program, and exploring another method family.

4.2 Skill Sources and Routing

ERS routes between two Skill sources. Task Skill comes from the expert experience in Section 3, providing useful advice for the current task. Failure Skill handles common engineering errors, such as dependencies, paths, array shapes, target encoding, and output formats. Lightweight data inspection is treated as a draft-stage check, not as a separate Skill source.

Each round selects only one full Skill source. The routing function is

$$\rho(b_r, d) = \begin{cases} S_{\text{task}}, & b_r = D, \\ S_{\text{error}}, & b_r = R, \\ S_{\text{task}}, & b_r = I. \end{cases}$$

Action	Goal	Skill source
Draft	Build task-skill anchored draft	Task Skill
Debug	Fix latest failure	Failure Skill
Improve	Raise validation score	Task Skill

Table 2: Action types and Skill routing in ERS.

Here d is the visible target-task description, and S_{task} is generated for d during the static experience stage. This design keeps the experience source aligned with the action goal. Draft and improve both use task-specific modeling experience, but with different intents: draft implements the Skill’s stable initial route, while improve makes bounded changes around a current best solution or explores a high-confidence alternative. Debug uses failure experience to handle concrete errors.

Single-source routing is a deliberate restriction. It prevents one round from mixing too many goals. If a round is meant to debug a shape error, it should not also import a new modeling recipe. If a round is meant to improve a stable solution, it should not spend most of the context on generic data checks. The restriction makes each plan easier to inspect. It also makes failures easier to attribute. When a debug round fails, the system knows that the chosen failure Skill and the latest error were the main inputs.

4.3 Separating Planning and Code

ERS splits each round into planning and implementation. The planning stage takes the task description, current memory, search intent, and routed Skill. It outputs a short plan contract:

$$p_r = P(x, M_r, i_r, \rho(b_r, d)).$$

The implementation stage generates a candidate program from the plan:

$$c_r = C(x, p_r).$$

The plan contract contains the round goal, data and submission constraints, the chosen method, the difference from the current best program, and main risks. This middle layer rewrites the Skill into concrete task actions. It may state which fields to inspect, which ID column to preserve, which validation strategy to use, or which model family to avoid in this round.

During draft actions, ERS can first run a read-only data check. This gives file structure, column names, label distributions, and a sample-

submission summary. If the check fails, the system records the cause and continues planning. The check summary is used as schema and resource evidence, not as the main modeling source. Thus the first draft is grounded in data evidence while still being anchored by the task-specific Skill.

The plan contract also limits unnecessary code churn. A coding agent may rewrite a working script when only a small fix is needed. ERS asks the plan to state what must stay fixed. In a repair round, this may be the current data loading path and the submission writer. In an improve round, this may be the validation split and the feature pipeline. The goal is not to freeze the code. The goal is to make each change local enough that feedback can be interpreted.

4.4 Branch Selection

ERS chooses the next branch from recent feedback. A draft action is preferred at the start of a task, after a missing file or schema uncertainty, or after a debug step destroys runnability. A debug action is preferred after a concrete error, such as an import failure, a shape mismatch, a missing column, or an invalid submission. An improve action is preferred after the program runs and produces a valid score.

The branch rule is intentionally simple. It does not try to predict the best model family directly. Instead, it chooses the kind of evidence the next round should use. This is useful in open AutoML because the same validation score can mean different things. A low score after a valid run may call for better modeling. A low score with row mismatch warnings may call for output-format debugging. A missing score after a crash should not trigger another model search. It should trigger error handling.

The memory state stores the evidence behind each branch decision. The recent run summary records the last error type and score trend. The method-category record stores whether recent attempts are still in the same family. The task memory stores stable facts such as label encoding, metric direction, validation split, and submission format. These facts are updated only when new execution evidence supports the update.

4.5 Novelty Constraints and Method Memory

The improve stage should reuse the current best solution, but it should not stay near one hypothesis

for too long. ERS uses two improve intents. Local improvement keeps the main body of the current best program and makes a few high-confidence changes. Alternative exploration uses the current best program as a reference and asks for a different method category.

Each plan contract records how the current round differs from the current best program or recent attempts. The record has three parts: what changed, what was kept for stability, and which method category was used. After execution, the system writes the method category, core components, and result reflection into memory M_{r+1} . This turns one run of feedback into in-task experience for the next round.

If recent improve rounds do not match or exceed the current best score, or if recent method categories repeat too often, ERS changes the next improve action to alternative exploration. Search then moves from small edits near the current best program to new modeling hypotheses. This forms an in-task experience evolution loop.

5 Experiments

The experiments study two questions. First, do cross-task Skills improve the quality of machine learning engineering agents? Second, can the proposed Experience-Evolving Meta-Agent (EEMA) combine experience selection and in-task evolution to obtain both strong task performance and reliable execution? We analyze both threshold-based outcomes, including medal counts and *Medal (%)*, and rank-sensitive outcomes, including *Grade (%)*. This lets us test whether a method only crosses medal thresholds or also improves the average competitive position of its submissions.

5.1 Setup

We use MLE-bench Lite as the main evaluation set. It contains 22 engineering tasks converted from public machine learning competitions. Each task asks an agent to generate a single-file solution. The program reads input from a given data directory and writes a submission file in the current working directory. The code generation stage only produces the solution file. Program execution, submission checks, and feedback collection are handled by an external evaluation sandbox. All systems use the same 12-hour sandbox budget.

We report three groups of metrics in Table 3. Gold, silver, and bronze counts measure the final

Model	Budget	Gold	Silver	Bronze	Medal (%) $\uparrow$	Grade (%) $\downarrow$	Success (%) $\uparrow$
<i>Baselines</i>							
MiniMax-M2.7	12h	4	4	2	45.5	44.5	64.8
Claude-Sonnet-4.6	12h	8	4	0	54.5	23.3	77.9
GPT-5.4-medium	12h	6	2	3	50.0	38.0	81.4
GPT-5.5-xhigh	12h	7	6	2	68.2	21.7	82.2
<i>Ablations</i>							
GPT-5.4-medium with Task Skill	12h	8	4	3	68.2	25.9	63.1
GPT-5.4-medium with Failure Skill	12h	6	4	1	50.0	34.9	88.9
GPT-5.4-medium with memory	12h	7	4	1	54.5	25.3	88.3
EEMA without Skills(GPT-5.4-medium)	12h	8	3	1	54.5	22.0	98.4
<i>Ours</i>							
EEMA(GPT-5.4-medium)	12h	7	6	4	77.3	20.1	96.0
EEMA(Claude-Sonnet-4.6)	12h	8	5	3	72.7	19.8	91.3

Table 3: Main comparison under a 12-hour sandbox budget. All rows represent agent systems rather than standalone language models. GPT rows use Codex; other model rows use Claude Code. *Grade (%)* is average normalized public-leaderboard rank, lower is better.

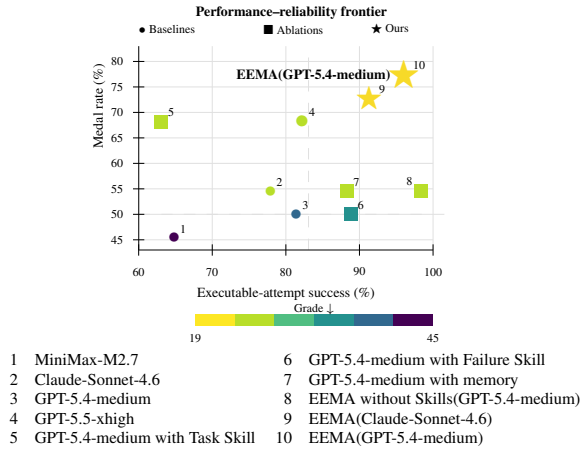


Figure 5: Performance-reliability frontier. Higher *Medal* and *Success* are preferred, while lower *Grade* is better and is encoded by marker color.

competitive level reached by the agent. *Medal (%)* is the percentage of tasks on which the agent obtains any medal. *Grade (%)* is the average normalized rank on the public leaderboard, so lower values indicate stronger submissions across the whole task set. *Success (%)* measures whether the agent can produce valid executable attempts during the search process. These metrics are complementary: medal rate reflects threshold-level competitive quality, grade reflects average ranking quality, and success rate reflects the stability of the engineering loop. Appendix C gives the formal definition of *Grade*.

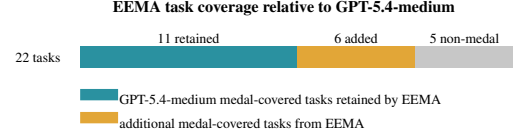


Figure 6: Task-level medal coverage decomposition for EEMA over the 22 MLE-bench Lite tasks.

Medal	Tasks
Gold	aerial-cactus; denoising-dirty-docs; insults; dogs-vs-cats; histopath-cancer; plant-pathology; tabular-dec-2021
Silver	aptos2019; leaf; nomad2018; random-pizza; spooky-author; text-normalization-en
Bronze	jigsaw-toxic; mlsp-birds; text-normalization-ru; whale-redux
N/A	dog-breed; nyc-taxi; ranzcr-catheter; siim-isic; tabular-may-2022

Table 4: Compact task-level medal outcomes for EEMA(GPT-5.4-medium), grouped from the full task-level table in the appendix.

5.2 Main Results

Table 3 and Figure 5 show that EEMA gives the best overall tradeoff under the same 12-hour budget. It obtains 17 medals out of 22 tasks, improving over GPT-5.4-medium by six medals and over GPT-5.5-xhigh by two medals. Its *Grade* also drops from 38.0% and 21.7% to 20.1%. EEMA also transfers effectively to Claude-Sonnet-4.6, achieving 16 medals (72.7%), a 19.8% *Grade*, and 91.3% *Success*. The frontier view makes this joint gain visible: both EEMA variants occupy the upper-right region while maintaining the best color-coded rank quality.

Figure 6 and Table 4 explain where the gain comes from. Relative to GPT-5.4-medium, EEMA keeps all 11 baseline medal-covered tasks and adds six more. The added coverage is not confined to one family: medals span image, text, tabular, and document-processing tasks, with the remaining five tasks left as N/A. This spread matters because MLE-bench Lite mixes modalities and scoring rules; gains concentrated in one recipe would be easier to dismiss than medals distributed across heterogeneous workflows.

5.3 Ablation Analysis

The ablations separate modeling experience, failure experience, memory, and the EEMA controller. Task Skills provide the largest direct quality gain for GPT-5.4-medium, raising the medal count from 11 to 15 and reducing *Grade* from 38.0% to 25.9%. Failure-prevention Skills mainly improve execution reliability, increasing *Success* from 81.4% to 88.9%. Runtime memory improves all three headline metrics over the GPT-5.4-medium baseline, but remains below full EEMA.

The no-skill controller isolates routing from external Skills. It reaches the highest *Success* rate, 98.4%, and a strong 22.0% *Grade*, but only 54.5% medal coverage. Full EEMA trades a small amount of success for much higher quality: 77.3% medal coverage and 20.1% *Grade*. The components are therefore complementary. Task Skills push stronger modeling ideas, Failure Skills keep the loop executable, and memory preserves what has been tried; EEMA routes these signals dynamically instead of appending every source to the prompt.

6 Conclusion

We studied how machine learning engineering agents can organize and evolve past experience during open-ended task solving. EEMA distills expert competition material and failure traces into compact Skills, then uses ERS to select, route, and update these Skills across multi-round solving. On MLE-bench Lite, task Skills raise medal coverage and average grade, failure-prevention Skills reduce runtime errors, and the full controller coordinates both to achieve 77.3% medal rate, 20.1% average grade, and 96.0% execution success. These results suggest that structured experience routing is a practical complement to model scaling for open AutoML.

Limitations

The current experiments mainly use MLE-bench Lite, which has a limited number of tasks. The conclusions should be checked on larger task sets. The dynamic-search evaluation is limited to the current EEMA setting and ablations in this paper; broader controller variants, larger task sets, and longer budgets remain future work. Skill effectiveness also depends on the instruction-following ability of the base model. Although the experience base uses leakage control, the quality of the original competition material can still affect the distilled Skills.

Ethical Considerations

This work uses public competition material and historical run traces to build the experience base. During construction, we exclude material from competitions that match evaluation tasks to avoid leakage risk. The system is meant to assist machine learning engineering. It does not target personal data re-identification or user profiling. Automated modeling systems may still amplify bias in source data. They may also produce models that look valid but lack enough validation. In real use, outputs should go through data compliance checks, task suitability checks, and human review.

Acknowledgments

This work is supported by the National Science and Technology Major Project (2023ZD0121403).

References

- Jun Shern Chan, Neil Chowdhury, Oliver Jaffe, James Aung, Dane Sherburn, Evan Mays, Giulio Starace, Kevin Liu, Leon Maksin, Tejal Patwardhan, Lilian Weng, and Aleksander Madry. 2025. MLE-bench: Evaluating machine learning agents on machine learning engineering. In *International Conference on Learning Representations*.
- Shangheng Du, Xiangchao Yan, Dengyang Jiang, Jiakang Yuan, Yusong Hu, Xin Li, Liang He, Bo Zhang, and Lei Bai. 2025. AutoMLGen: Navigating fine-grained optimization for coding agents. *arXiv preprint arXiv:2510.08511*.
- Matthias Feurer, Aaron Klein, Katharina Eggenberger, Jost Springenberg, Manuel Blum, and Frank Hutter. 2015. Efficient and robust automated machine learning. *Advances in Neural Information Processing Systems*, 28.

- Che Jiang, Jincheng Zhong, Yu Fu, Kai Tian, Junlin Yang, Kaikai Zhao, Yuchong Wang, Tianwei Luo, Weizhi Wang, Yuxin Zuo, Guoli Jia, Xingtai Lv, Dianqiao Lei, Sihang Zeng, Yuru Wang, Zhenzhao Yuan, Xinwei Long, Ermo Hua, Can Ren, and 8 others. 2026. [Self-improving agents in the era of experience: A survey of self- to meta-evolution](#). *Open-Review Archive*.
- Zhengyao Jiang, Dominik Schmidt, Dhruv Srikanth, Dixing Xu, Ian Kaplan, Deniss Jacenko, and Yuxiang Wu. 2025. AIDE: AI-driven exploration in the space of code. *arXiv preprint arXiv:2502.13138*.
- Stepan Kulibaba, Artem Dzhililov, Roman Pakhomov, Oleg Svidchenko, Alexander Gasnikov, and Aleksei Shpilman. 2025. KompeteAI: Accelerated autonomous multi-agent system for end-to-end pipeline generation for machine learning problems. *arXiv preprint arXiv:2508.10177*.
- Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich K ttler, Mike Lewis, Wen-tau Yih, Tim Rock-t schel, Sebastian Riedel, and Douwe Kiela. 2020. Retrieval-augmented generation for knowledge-intensive NLP tasks. In *Advances in Neural Information Processing Systems*, volume 33, pages 9459–9474.
- Hongyi Liu, Haoyan Yang, Tao Jiang, Bo Tang, Feiyu Xiong, and Zhiyu Li. 2026. SkillsVote: Lifecycle governance of agent skills from collection, recommendation to evolution. *arXiv preprint arXiv:2605.18401v1*.
- Zexi Liu, Yuzhu Cai, Xinyu Zhu, Yujie Zheng, Runkun Chen, Ying Wen, Yanfeng Wang, Weinan E, and Siheng Chen. 2025. ML-Master: Towards AI-for-AI via integration of exploration and reasoning. *arXiv preprint arXiv:2506.16499*.
- Alisia Lupidi, Bhavul Gauri, Thomas Simon Foster, Bassel Al Omari, Despoina Magka, Alberto Pepe, Alexis Audran-Reiss, Muna Aghamelu, Nicolas Baldwin, Lucia Cipolina-Kun, Jean-Christophe Gagnon-Audet, Chee Hau Leow, Sandra Lefdal, Hossam Mossalam, Abhinav Moudgil, Saba Nazir, Emanuel Tewolde, Isabel Urrego, Jordi Armengol Estape, and 18 others. 2026. AIRS-Bench: A suite of tasks for frontier AI research science agents. *arXiv preprint arXiv:2602.06855*.
- Jaehyun Nam, Jinsung Yoon, Jiefeng Chen, Jinwoo Shin, Sercan O. Arik, and Tomas Pfister. 2025. MLE-STAR: Machine learning engineering agent via search and targeted refinement. *arXiv preprint arXiv:2506.15692*.
- Randal S Olson, Nathan Bartley, Ryan J Urbanowicz, and Jason H Moore. 2016. Evaluation of a tree-based pipeline optimization tool for automating data science. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 485–492.
- Rushi Qiang, Yuchen Zhuang, Yinghao Li, Dingu Sagar V K, Rongzhi Zhang, Changhao Li, Ian Shu-Hei Wong, Sherry Yang, Percy Liang, Chao Zhang, and Bo Dai. 2025a. MLE-Dojo: Interactive environments for empowering LLM agents in machine learning engineering. *arXiv preprint arXiv:2505.07782*.
- Rushi Qiang, Yuchen Zhuang, Anikait Singh, Percy Liang, Chao Zhang, Sherry Yang, and Bo Dai. 2025b. MLE-Smith: Scaling MLE tasks with automated multi-agent pipeline. *arXiv preprint arXiv:2510.07307*.
- Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36:8634–8652.
- Chandan Singh, Yan Shuo Tan, Weijia Xu, Zelalem Gero, Weiwei Yang, Michel Galley, and Jianfeng Gao. 2026. Agentic-iModels: Evolving agentic interpretability tools via AutoResearch. *arXiv preprint arXiv:2605.03808*.
- Chris Thornton, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. 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.
- Edan Toledo, Karen Hambardzumyan, Martin Josifoski, Rishi Hazra, Nicolas Baldwin, Alexis Audran-Reiss, Michael Kuchnik, Despoina Magka, Minqi Jiang, Alisia Maria Lupidi, Andrei Lupu, Roberta Raileanu, Kelvin Niu, Tatiana Shavrina, Jean-Christophe Gagnon-Audet, Michael Shvartsman, Shagun Sodhani, Alexander H. Miller, Abhishek Charnalia, and 6 others. 2025. AI research agents for machine learning: Search, exploration, and generalization in MLE-bench. *arXiv preprint arXiv:2507.02554*.
- Joaquin Vanschoren, Jan N. van Rijn, Bernd Bischl, and Luis Torgo. 2014. OpenML: Networked science in machine learning. *SIGKDD Explorations*, 15(2):49–60.
- Shijie Wang, Pengfei Li, Yikun Fu, Kaifeng Liu, Fangyuan Li, Yang Liu, Xiaowei Sun, Zonglin Li, Siyao Zhao, Jian Zhao, Kai Tian, Dong Li, Junqi Gao, Yutong Zhang, Yiqun Chen, Yuqiang Li, Zoe Li, Weinan Zhang, Peng Ye, and 5 others. 2026a. MARTI-MARS²: Scaling multi-agent self-search via reinforcement learning for code generation. *arXiv preprint arXiv:2602.07848*.
- Yuru Wang, Lejun Cheng, Yuxin Zuo, Sihang Zeng, Bingxiang He, Che Jiang, Junlin Yang, Yuchong Wang, Kaikai Zhao, Weifeng Huang, Kai Tian, Zhenzhao Yuan, Jincheng Zhong, Weizhi Wang, Ning Ding, Bowen Zhou, and Kaiyan Zhang. 2026b.

NatureBench: Can coding agents match the published SOTA of nature-family papers? *arXiv preprint arXiv:2606.24530*.

Minghao Yan, Bo Peng, Benjamin Coleman, Ziqi Chen, Zhouhang Xie, Shuo Chen, Zhankui He, Noveen Sachdeva, Weili Wang, Ed H. Chi, Shivaram Venkataraman, Wang-Cheng Kang, Derek Zhiyuan Cheng, and Beidou Wang. 2026. PACEvolve++: Improving test-time learning for evolutionary search agents. *arXiv preprint arXiv:2605.07039*.

Junlin Yang, Che Jiang, Yu Fu, Tianwei Luo, Can Ren, Weizhi Wang, Kaikai Zhao, Hongyi Liu, Yuxin Zuo, Yuru Wang, Yuchen Fan, Kai Tian, Zhenzhao Yuan, Xiaojian Lin, Li Sheng, Rushi Qiang, Guoli Jia, Xingtai Lv, Ermo Hua, and 5 others. 2026. Frontis-MA1: Training an AI4AI model towards recursive self-improvement in machine learning engineering. *arXiv preprint arXiv:2607.28568*.

Sherry Yang, Joy He-Yueya, and Percy Liang. 2025a. Reinforcement learning for machine learning engineering agents. *arXiv preprint arXiv:2509.01684*.

Xu Yang, Xiao Yang, Shikai Fang, Yifei Zhang, Jian Wang, Bowen Xian, Qizheng Li, Jingyuan Li, Minrui Xu, Yuante Li, Haoran Pan, Yuge Zhang, Weiqing Liu, Yelong Shen, Weizhu Chen, and Jiang Bian. 2025b. R&D-Agent: An LLM-agent framework towards autonomous data science. *arXiv preprint arXiv:2505.14738*.

A Runtime Memory Record

ERS keeps a short memory after each run. The memory is used for later branch choice and planning. It is not a full transcript. It keeps only facts that can change the next action. Table 5 shows the fields used in our design.

The relative-change field is useful in practice. Without it, an agent may keep trying small edits that look new in code but repeat the same modeling idea. The method category gives a simple signal for this case. If recent improve rounds all use the same family and the score does not improve, ERS can move from local improvement to alternative exploration.

B Additional Result Analysis

The main table shows that EEMA improves over strong base agents rather than merely benefiting from a larger underlying model. GPT-5.5-xhigh (Codex) is the strongest large-LLM baseline with a 68.2% medal rate and a 21.7% grade, while EEMA reaches a 77.3% medal rate and a 20.1% grade. This gap is important because EEMA does not simply replace the base agent with a stronger one; it changes how experience is selected and

used during the run. The result suggests that experience evolution can add value on top of model scaling.

The comparison between EEMA without Skills and full EEMA separates controller stability from experience quality. EEMA without Skills has the highest success rate, 98.4%, but only a 54.5% medal rate. This means that the controller can keep the loop executable, but stability alone does not provide enough modeling signal to reach high medal coverage. Full EEMA reduces success only slightly to 96.0%, while raising the medal rate to 77.3%. This is the desired tradeoff: the system keeps nearly the same execution reliability while substantially improving solution quality.

Together, these results support the routing design. Task Skills are useful when the system needs to improve a valid solution, failure-prevention Skills are useful when the system needs to debug or avoid invalid runs, and memory helps decide when repeated local changes are no longer useful. Full EEMA obtains the best medal rate because it does not apply every experience source uniformly. It routes the source according to the current search state.

C Grade Metric

For each task, we convert the final submission score into a normalized public-leaderboard rank. Let $L_t = (s_{t,1}, \dots, s_{t,n_t})$ be the ordered public leaderboard scores for task t , where n_t is the number of public teams. The score direction is inferred from the leaderboard ordering. For a submission score $\hat{s}_t$, we compute its rank as

$$r_t = 1 + |\{i : s_{t,i} \succ \hat{s}_t\}|,$$

where $s_{t,i} \succ \hat{s}_t$ means that the public score is strictly better than the submitted score under the task metric. We then normalize the rank by the number of teams:

$$g_t = \frac{r_t}{n_t}.$$

Lower values are better. A top-ranked submission has a grade close to zero, while a low-ranked or invalid submission is close to one. If a run does not produce a scored submission, we assign $g_t = 1$ for aggregation. The reported *Grade (%)* is $100 \times |T|^{-1} \sum_{t \in T} g_t$ over the evaluation tasks.

D Failure Modes

The failure-prevention Skills are based on repeated engineering failures. Table 6 groups common failures into seven contracts. The groups are broad, but they are concrete enough to guide planning.

These contracts explain why failure Skills reduce errors without always improving medals. They target validity first. They make sure the solution can run, score, and submit. After that, task-level Skills can focus on model quality. ERS separates these two roles so that a debug round does not turn into a new modeling experiment.

E Sandbox and Iteration Protocol

All runs use a 12-hour sandbox budget per task. The sandbox matches the execution boundary in the task prompt: 6 CPU cores, 200 GB RAM, 16 GB shared memory, and one 24 GB GPU for GPU-assigned tasks. The code agent writes a single-file solution, and the evaluation driver executes it, checks the submission file, records failures and scores, and returns feedback for later rounds.

During search, we do not use feedback from the official MLE-bench Lite test set. Following the MLE-bench splitting logic, we split each task’s provided training data into an internal training split and an internal evaluation split, preserving the original file structure, modality, target schema, and submission format as closely as possible. Candidate programs are trained on the internal training split and scored on the internal evaluation split throughout the 12-hour run.

After the budget expires, we select the valid candidate with the best internal score and submit that code for final evaluation on the original MLE-bench Lite test set. The results in Table 3 and Tables 7–16 report this final test-set evaluation, not the internal search score.

F Full Task-Level Results

Tables 7–16 report the task-level medal outcomes corresponding to Table 3. Only the task name and final medal status are shown. Missing medals are shown as N/A.

G Baseline Experiment Prompt

This section shows the baseline prompt used to ask the code agent for a single-file Kaggle solution. The concrete task metadata and task description are replaced by placeholders because the ap-

pendix focuses on the prompt structure rather than one full benchmark instance.

The full prompt listing is shown in Table 17.

H Task-Level Skill Synthesis Prompt

This section gives the prompt used to synthesize a task-specific Skill from the cleaned task description and task-family playbooks. It illustrates how the system asks for modeling guidance while excluding leakage-prone sources, hidden evaluation facts, and generic engineering checks handled by other Skills.

The full synthesis prompt is shown in Table 18.

I Task-Level Skill Example

This section provides a compact example of a generated task-level Skill for the MLSP 2013 bird classification task. The example keeps the task-specific modeling priorities and replaces later detailed upgrade lists with explicit omissions to keep the appendix readable.

The full example is shown in Table 19.

J Failure Prevention Skill Example

This section shows the failure-prevention Skill used as an engineering contract for planning and coding. The retained text highlights the contract structure, while detailed requirements are summarized with placeholders.

The full example is shown in Table 20.

Field	Content
Method summary	Main model family, feature pipeline, validation split, and output writer.
Result reflection	Whether the run failed, improved, or exposed a new task fact.
Method category	A short tag such as linear model, tree ensemble, CNN, transformer, or post-processing.
Relative change	What changed from the current best program and what stayed fixed.

Table 5: Runtime memory fields used by ERS.

Contract	Typical failure	Preventive check
Environment and API	Missing package, changed argument, unsupported device.	Import libraries early in the generated code and use conservative API calls.
Data schema	Wrong column name, wrong path, unexpected file extension.	Read file lists and columns before feature code.
Feature and target	Label encoding mismatch, leakage from test columns.	Build target mapping from train only and keep test columns separate.
Validation split	Split leaks groups or changes class balance.	Match split to task type, group fields, and metric.
Array and batch	Shape mismatch, dtype mismatch, bad batch collation.	Print or assert key shapes before training.
Modality boundary	Treating audio as tabular, or image meta-data as labels.	Identify modality before choosing a model family.
Output schema	Wrong row count, wrong ID order, extra columns.	Copy sample submission and fill only prediction columns.

Table 6: Failure-prevention contracts used to build failure Skills.

Task	Medal
aerial-cactus-identification	N/A
aptos2019-blindness-detection	Silver
denoising-dirty-documents	Silver
detecting-insults-in-social-commentary	Gold
dog-breed-identification	N/A
dogs-vs-cats-redux-kernels-edition	Gold
histopathologic-cancer-detection	Gold
jigsaw-toxic-comment-classification-challenge	N/A
leaf-classification	N/A
mlsp-2013-birds	N/A
new-york-city-taxi-fare-prediction	N/A
nomad2018-predict-transparent-conductors	Silver
plant-pathology-2020-fgvc7	Gold
random-acts-of-pizza	Bronze
ranzcr-clip-catheter-line-classification	N/A
siim-isic-melanoma-classification	N/A
spooky-author-identification	Bronze
tabular-playground-series-dec-2021	N/A
tabular-playground-series-may-2022	N/A
text-normalization-challenge-english-language	Silver
text-normalization-challenge-russian-language	N/A
the-icml-2013-whale-challenge-right-whale-redux	N/A

Table 7: Task-level medal outcomes for MiniMax-M2.7.

Task	Medal
aerial-cactus-identification	N/A
aptos2019-blindness-detection	N/A
denoising-dirty-documents	Gold
detecting-insults-in-social-commentary	Gold
dog-breed-identification	N/A
dogs-vs-cats-redux-kernels-edition	Gold
histopathologic-cancer-detection	Gold
jigsaw-toxic-comment-classification-challenge	Gold
leaf-classification	N/A
mlsp-2013-birds	N/A
new-york-city-taxi-fare-prediction	N/A
nomad2018-predict-transparent-conductors	Gold
plant-pathology-2020-fgvc7	Gold
random-acts-of-pizza	N/A
ranzcr-clip-catheter-line-classification	N/A
siim-isic-melanoma-classification	N/A
spooky-author-identification	Silver
tabular-playground-series-dec-2021	Gold
tabular-playground-series-may-2022	N/A
text-normalization-challenge-english-language	Silver
text-normalization-challenge-russian-language	Silver
the-icml-2013-whale-challenge-right-whale-redux	Silver

Table 8: Task-level medal outcomes for Claude-Sonnet-4.6.

Task	Medal
aerial-cactus-identification	N/A
aptos2019-blindness-detection	N/A
denoising-dirty-documents	Gold
detecting-insults-in-social-commentary	Gold
dog-breed-identification	N/A
dogs-vs-cats-redux-kernels-edition	Gold
histopathologic-cancer-detection	Gold
jigsaw-toxic-comment-classification-challenge	N/A
leaf-classification	Bronze
mlsp-2013-birds	Bronze
new-york-city-taxi-fare-prediction	N/A
nomad2018-predict-transparent-conductors	Gold
plant-pathology-2020-fgvc7	Gold
random-acts-of-pizza	N/A
ranzcr-clip-catheter-line-classification	N/A
siim-isic-melanoma-classification	N/A
spooky-author-identification	N/A
tabular-playground-series-dec-2021	N/A
tabular-playground-series-may-2022	N/A
text-normalization-challenge-english-language	Silver
text-normalization-challenge-russian-language	Bronze
the-icml-2013-whale-challenge-right-whale-redux	Silver

Table 9: Task-level medal outcomes for GPT-5.4-medium.

Task	Medal
aerial-cactus-identification	Gold
aptos2019-blindness-detection	Silver
denoising-dirty-documents	Gold
detecting-insults-in-social-commentary	Gold
dog-breed-identification	N/A
dogs-vs-cats-redux-kernels-edition	Gold
histopathologic-cancer-detection	Gold
jigsaw-toxic-comment-classification-challenge	N/A
leaf-classification	Bronze
mlsp-2013-birds	Silver
new-york-city-taxi-fare-prediction	N/A
nomad2018-predict-transparent-conductors	Gold
plant-pathology-2020-fgvc7	Silver
random-acts-of-pizza	Silver
ranzcr-clip-catheter-line-classification	N/A
siim-isic-melanoma-classification	N/A
spooky-author-identification	N/A
tabular-playground-series-dec-2021	Gold
tabular-playground-series-may-2022	N/A
text-normalization-challenge-english-language	Silver
text-normalization-challenge-russian-language	Bronze
the-icml-2013-whale-challenge-right-whale-redux	Silver

Table 10: Task-level medal outcomes for GPT-5.5-xhigh.

Task	Medal
aerial-cactus-identification	Gold
aptos2019-blindness-detection	Bronze
denoising-dirty-documents	Gold
detecting-insults-in-social-commentary	Gold
dog-breed-identification	N/A
dogs-vs-cats-redux-kernels-edition	Gold
histopathologic-cancer-detection	Gold
jigsaw-toxic-comment-classification-challenge	N/A
leaf-classification	Silver
mlsp-2013-birds	Bronze
new-york-city-taxi-fare-prediction	N/A
nomad2018-predict-transparent-conductors	Gold
plant-pathology-2020-fgvc7	Gold
random-acts-of-pizza	N/A
ranzcr-clip-catheter-line-classification	N/A
siim-isic-melanoma-classification	N/A
spooky-author-identification	Silver
tabular-playground-series-dec-2021	Gold
tabular-playground-series-may-2022	N/A
text-normalization-challenge-english-language	Silver
text-normalization-challenge-russian-language	Bronze
the-icml-2013-whale-challenge-right-whale-redux	Silver

Table 11: Task-level medal outcomes for GPT-5.4-medium with Task Skill.

Task	Medal
aerial-cactus-identification	N/A
aptos2019-blindness-detection	N/A
denoising-dirty-documents	Gold
detecting-insults-in-social-commentary	Gold
dog-breed-identification	N/A
dogs-vs-cats-redux-kernels-edition	Gold
histopathologic-cancer-detection	Gold
jigsaw-toxic-comment-classification-challenge	N/A
leaf-classification	N/A
mlsp-2013-birds	Bronze
new-york-city-taxi-fare-prediction	N/A
nomad2018-predict-transparent-conductors	Silver
plant-pathology-2020-fgvc7	Gold
random-acts-of-pizza	Silver
ranzcr-clip-catheter-line-classification	N/A
siim-isic-melanoma-classification	N/A
spooky-author-identification	N/A
tabular-playground-series-dec-2021	N/A
tabular-playground-series-may-2022	N/A
text-normalization-challenge-english-language	Silver
text-normalization-challenge-russian-language	Silver
the-icml-2013-whale-challenge-right-whale-redux	Gold

Table 12: Task-level medal outcomes for GPT-5.4-medium with Failure Skill.

Task	Medal
aerial-cactus-identification	Gold
aptos2019-blindness-detection	Gold
denoising-dirty-documents	Gold
detecting-insults-in-social-commentary	Gold
dog-breed-identification	N/A
dogs-vs-cats-redux-kernels-edition	Gold
histopathologic-cancer-detection	Gold
jigsaw-toxic-comment-classification-challenge	N/A
leaf-classification	N/A
mlsp-2013-birds	Silver
new-york-city-taxi-fare-prediction	N/A
nomad2018-predict-transparent-conductors	Silver
plant-pathology-2020-fgvc7	N/A
random-acts-of-pizza	N/A
ranzcr-clip-catheter-line-classification	N/A
siim-isic-melanoma-classification	N/A
spooky-author-identification	Bronze
tabular-playground-series-dec-2021	N/A
tabular-playground-series-may-2022	N/A
text-normalization-challenge-english-language	Silver
text-normalization-challenge-russian-language	Silver
the-icml-2013-whale-challenge-right-whale-redux	Gold

Table 13: Task-level medal outcomes for GPT-5.4-medium with memory.

Task	Medal
aerial-cactus-identification	N/A
aptos2019-blindness-detection	N/A
denoising-dirty-documents	Gold
detecting-insults-in-social-commentary	Gold
dog-breed-identification	N/A
dogs-vs-cats-redux-kernels-edition	Gold
histopathologic-cancer-detection	Gold
jigsaw-toxic-comment-classification-challenge	N/A
leaf-classification	N/A
mlsp-2013-birds	Gold
new-york-city-taxi-fare-prediction	N/A
nomad2018-predict-transparent-conductors	Gold
plant-pathology-2020-fgvc7	Gold
random-acts-of-pizza	Silver
ranzcr-clip-catheter-line-classification	N/A
siim-isic-melanoma-classification	N/A
spooky-author-identification	N/A
tabular-playground-series-dec-2021	Gold
tabular-playground-series-may-2022	N/A
text-normalization-challenge-english-language	Silver
text-normalization-challenge-russian-language	Bronze
the-icml-2013-whale-challenge-right-whale-redux	Silver

Table 14: Task-level medal outcomes for EEMA without Skills(GPT-5.4-medium).

Task	Medal
aerial-cactus-identification	Gold
aptos2019-blindness-detection	Silver
denoising-dirty-documents	Gold
detecting-insults-in-social-commentary	Gold
dog-breed-identification	N/A
dogs-vs-cats-redux-kernels-edition	Gold
histopathologic-cancer-detection	Gold
jigsaw-toxic-comment-classification-challenge	Bronze
leaf-classification	Silver
mlsp-2013-birds	Bronze
new-york-city-taxi-fare-prediction	N/A
nomad2018-predict-transparent-conductors	Silver
plant-pathology-2020-fgvc7	Gold
random-acts-of-pizza	Silver
ranzcr-clip-catheter-line-classification	N/A
siim-isic-melanoma-classification	N/A
spooky-author-identification	Silver
tabular-playground-series-dec-2021	Gold
tabular-playground-series-may-2022	N/A
text-normalization-challenge-english-language	Silver
text-normalization-challenge-russian-language	Bronze
the-icml-2013-whale-challenge-right-whale-redux	Bronze

Table 15: Task-level medal outcomes for EEMA(GPT-5.4-medium).

Task	Medal
aerial-cactus-identification	Gold
aptos2019-blindness-detection	Bronze
denoising-dirty-documents	Gold
detecting-insults-in-social-commentary	Gold
dog-breed-identification	N/A
dogs-vs-cats-redux-kernels-edition	Gold
histopathologic-cancer-detection	Gold
jigsaw-toxic-comment-classification-challenge	Silver
leaf-classification	Bronze
mlsp-2013-birds	N/A
new-york-city-taxi-fare-prediction	N/A
nomad2018-predict-transparent-conductors	Silver
plant-pathology-2020-fgvc7	Gold
random-acts-of-pizza	Silver
ranzcr-clip-catheter-line-classification	N/A
siim-isic-melanoma-classification	N/A
spooky-author-identification	Silver
tabular-playground-series-dec-2021	Gold
tabular-playground-series-may-2022	N/A
text-normalization-challenge-english-language	Silver
text-normalization-challenge-russian-language	Bronze
the-icml-2013-whale-challenge-right-whale-redux	Gold

Table 16: Task-level medal outcomes for EEMA(Claude-Sonnet-4.6).

BASELINE EXPERIMENT PROMPT

[SYSTEM INSTRUCTIONS]

You are a top-ranked Kaggle grandmaster with extensive competition experience. Your objective is to solve a Kaggle competition, with the goal of maximizing the Position Score (Your rank in the leaderboard) in limited steps. You must use Machine Learning/Deep Learning/Computer Vision/NLP/etc. methods to solve the problem, the score of random guess or without any ML/DL/CV/NLP methods will be cancelled finally. You are likely to train models according to specific competition requirements.

CRITICAL INSTRUCTIONS - READ CAREFULLY:

1. DO NOT execute any code - do not run Python scripts or shell commands
2. DO NOT run the solution yourself
3. Your ONLY task is to create a file named `solution.py` containing the complete solution code
4. Create `solution.py` by writing it to the current working directory
5. The code will be executed later in a separate sandbox environment
6. After creating the file, simply confirm its creation - DO NOT RUN IT

Code Requirements:

- The code should be a single-file python program that is self-contained and can be executed as-is
- No parts of the code should be skipped, don't terminate the script before finishing the code
- All input data is located in the base path specified by the DATA_DIR environment variable. Always access the data path using `os.environ.get("DATA_DIR")` and never hardcode absolute paths.
- **Save test predictions to `submission.csv` in `./` directory as specified in the task description**
- The code can use GPU and PyTorch/CUDA for faster training if needed

Remember: CREATE the solution.py file, DO NOT EXECUTE it.

[USER TASK]

[SYSTEM]

You must use Machine Learning/Deep Learning/Computer Vision/NLP/etc. methods to solve the problem, the score of random guess or without any ML/DL/CV/NLP methods will be cancelled finally. You are operating in a Python environment where the following machine learning-related packages are preinstalled: torch, tensorflow, scikit-learn, numpy, pandas, xgboost, lightgbm, catboost, transformers, datasets, tokenizers, huggingface-hub, accelerate, torchvision, torchaudio, timm, torch-geometric, scipy, optuna, bayesian-optimization, shap, spacy, nltk, sentencepiece, opencv-python, scikit-image, pillow, albumentations, librosa, soundfile, speechbrain, openai-whisper, sentence-transformers, tiktoken, einops, safetensors, keras, ultralytics. If you need to, feel free to use additional libraries that fit the problem. For neural networks, we suggest using PyTorch rather than TensorFlow. You are running in a sandboxed environment with the following resources:

- CPU: 6 cores
- System Memory (RAM): 200 GB
- Shared Memory (shm-size): 16 GB
- GPU Memory (VRAM): 24 GB

Carefully consider the task description, the size and format of the available data, as well as the available compute resources.

[USER]

Output Instructions:

Your response should be a brief outline/sketch of your proposed solution in natural language, followed by a single markdown code block (wrapped in ```) which implements this solution. There should be no additional headings or text in your response. Just natural language text followed by a newline and then the markdown code block. The code should be a single-file python program that is self-contained and can be executed as-is. No parts of the code should be skipped. Don't terminate the script before finishing the code. All input data is located in the base path specified by the DATA_DIR environment variable. Always access the data path using `os.environ.get("DATA_DIR")` and never hardcode absolute paths. **SAVE PREDICTIONS ON THE PROVIDED UNLABELED TEST DATA IN A `submission.csv` FILE IN THE `./` DIRECTORY. ** Your metric score depends on generating a valid `submission.csv` file at the correct location.

CONSTRAINTS:

- Use `torch.optim.AdamW` (the recommended optimizer) instead of the deprecated `AdamW` from `transformers`.
- In LightGBM, deprecated arguments like `early_stopping_rounds` must be replaced with callbacks (e.g., `lightgbm.early_stopping(stopping_rounds=...)`), and training controls (e.g., `verbosity`) should also use callbacks, while model parameters such as `class_weight` must be set during initialization rather than passed to `fit()`.
- In recent versions of Hugging Face Transformers, use `eval_strategy` instead of the deprecated `evaluation_strategy`, and do not pass `class_weight` into `TrainingArguments`; instead, handle class weighting in the loss computation.
- If you use albumentations: 1. For `RandomResizedCrop` and similar transforms (including transforms with a `size` field, geometric transforms, crop transforms), pass `size=(H, W)` as a tuple instead of a single integer or incompatible positional arguments. 2. Avoid guessing transform names or parameters such as `Cutout`, `Flip`, `RandomHorizontalFlip` and `RandomErasing`. Stick to verified stable APIs.

[METADATA]

Task: <task-slug>
Resource Type: <CPU|GPU>
Data Directory: <DATA_DIR>
Evaluation Metric Range: <min, max>
Higher is Better: <true|false>

[DATA DESCRIPTION]

<Concise file manifest and schema summary supplied by the benchmark environment.>
<Include only the fields needed to understand input files, sample submission, train/test schema, and available modalities.>

[TASK DESCRIPTION]

<Task overview, evaluation metric, submission format, and dataset description supplied by the benchmark environment.>
<The concrete competition text is omitted here to keep the appendix compact.>

Table 17: Baseline experiment prompt.

TASK-LEVEL SKILL SYNTHESIS PROMPT

You are an expert Kaggle Grandmaster-style skill writer and ML strategy designer.

Your task is to build one high-performance MLE-bench Lite skill for the specified target task:

<task>

You are working in a Markdown knowledge-base repository. This repository is the authority for task-specific guidance. Use general ML expertise only to synthesize, prioritize, and fill standard implementation details when repository materials do not provide conflicting guidance. DO NOT use the internet.

Repository structure:

- system.txt Shared execution instructions and environment assumptions for generated Kaggle solutions.
- descriptions/cleaned_<kaggle-slug>.txt Cleaned task descriptions.
- task-family-playbooks/Task-family playbooks distilled from leakage-filtered solution libraries, named by modality and task family.
- OUTPUTS/Destination for generated task-level skills.

Output: Create exactly one file:

./OUTPUTS/SKILL_<task>.md

If OUTPUTS/does not exist, create it. If ./OUTPUTS/SKILL_<task>.md already exists, overwrite it with the new final version.

Core objective: The generated skill must guide a strong downstream code-generation agent to write a complete Kaggle solution script for <task>, with the goal of achieving MLE-bench Lite medal-level or gold-level performance under the documented environment and time budget.

This skill is not a generic explanation. It is a high-density tactical modeling guide for a strong downstream agent. The downstream agent already knows generic ML and can code well. The skill should provide performance-relevant decisions, task-specific modeling priorities, strong model choices, validation/metric alignment, and round-by-round upgrade guidance.

Strict boundaries:

- Read and respect system.txt as hard boundary material for execution constraints and environment assumptions.
- Do not include runtime/error/checklist details that are handled elsewhere.
- Do not focus on path handling, API compatibility, submission.csv mechanics, package deprecation, defensive coding, or file I/O guardrails unless they directly affect modeling or metric optimization.
- Do not recommend external datasets, leaks, private labels, manual labels, or unavailable resources as the default plan.
- Do not mention internal baseline results, prior agent performance, score history, medals from internal runs, or hidden evaluation observations.
- Do not include source file paths, atom IDs, audit IDs, or raw retrieval/debug artifacts in the final skill.
- Do not copy long snippets from source documents. Synthesize.

Information priority:

1. descriptions/cleaned_<task>.txt Read this first. This is the target MLE-bench Lite task definition and should dominate all decisions.
2. system.txt Hard constraints and environment boundary. Use only to keep the strategy feasible.
3. task-family-playbooks/<task-family>.md or task-family-playbooks/<M>_<T>.md High-priority task-family playbooks. Infer relevant task families from the target description and read all matching playbooks.
4. Adjacent task-family playbooks Read adjacent families only when they plausibly improve modeling choices for the target task.
5. Other already generated OUTPUTS/SKILL_*.md Low-priority style/reference examples only. Do not copy task-specific advice from unrelated tasks.

Task-family selection: Infer the task family or families from descriptions/cleaned_<task>.txt. You may read multiple relevant task-family playbooks. Examples:

- [Task-family examples omitted for brevity.]
- Prioritize the family documents that best match the target task, metric, data shape, and submission target.

Important strategic stance: This skill is for a strong downstream code-generation agent. Do not recommend weak/simple baselines as the final intended strategy unless the task genuinely rewards them. The skill should push toward the strongest feasible solution within system.txt budget and environment.

However, the strategy must be task-fit first:

- First choose the approach that best matches the task description, metric, data modality, data size, and target format.
- Then choose the strongest feasible model family.
- Prefer powerful current models/backbones/ensembles when they are appropriate and safe under the budget.
- Balance strong first implementation with multi-round exploration.
- A first-round plan can be robust and not overcomplicated, but it should still be a serious medal-oriented route, not a toy baseline.

Reading procedure:

1. Read system.txt.
2. Read descriptions/cleaned_<task>.txt.
3. Infer likely task family/families, metric, prediction target, data modalities, and likely score levers.
4. Inspect task-family-playbooks/and read all relevant playbooks.
5. Optionally inspect nearby generated OUTPUTS/SKILL_*.md only for style consistency, not for content transfer.
6. Synthesize a final skill specifically for <task>.

Before writing the final file, reason through:

- What is this task really asking the solver to predict?
- What metric is used, and what does it imply for training/inference/postprocessing?
- What is the strongest feasible model route under system.txt?
- What first implementation gives a serious high-performing starting point?
- What upgrades are most likely to improve score across later rounds?
- What validation strategy is most trustworthy?
- What tricks should be avoided, delayed, or treated as risky?
- Which repository guidance is generic and should be discarded for this task?

Final markdown structure: Use exactly this structure.

Skill: <task>

```

## 1. Task-Specific Reading
- Identify the real task type, target structure, metric implications, and modality.
- Mention only details that affect modeling choices.
- State the dominant score levers.

## 2. Highest-Expected-Score Strategy
- Describe the strongest feasible route for this task under system.txt.
- Prefer high-performing, modern, task-appropriate models.
- Explain how model family, preprocessing/features, validation, metric optimization, and inference fit together.
- This section should tell the downstream agent what a medal/gold-oriented solution should converge toward.

## 3. Strong First Implementation Plan
- Give a concrete first-round implementation strategy that can be coded in one complete solution script.
- It should be strong enough to matter, not a toy baseline.
- Include model choice, important preprocessing/features, training target/loss, validation, and inference/postprocessing.
- Keep it feasible under the documented environment and budget.

## 4. High-ROI Upgrades Across Rounds
- Round 2:
- Round 3:
- Late round:
- Make upgrades ordered by expected leaderboard gain per implementation risk.
- Include stronger models, resolution/feature upgrades, ensembling, pseudo-labeling, stacking, TTA, thresholding, or postprocessing only when appropriate.

## 5. Validation and Metric Optimization
- Explain the validation split strategy and why.
- Explain the exact metric-aligned optimization behavior.
- Mention OOF usage, threshold tuning, calibration, target transforms, rank/probability outputs, aggregation, or clipping only when relevant.
- Emphasize what to trust when local CV and leaderboard may diverge.

## 6. Model, Feature, and Preprocessing Priorities
- List the highest-value model/features/preprocessing choices for this task.
- Be specific to the task.
- Prefer actionable priorities over generic statements.

## 7. Avoid or Delay
- List wrong routes, overcomplicated first-round choices, risky tricks, leakage-prone validation, metric-mismatched outputs, and external-data dependence.
- Include what to delay until after a stable strong baseline exists.

Length and style:
- Aim for around 2000 words if the task is complex.
- Do not exceed 3000 words.
- Keep information density high.
- Use concise, imperative bullets.
- Avoid long prose.
- Do not include code unless a tiny pseudocode-like formula or metric note is essential.
- Do not include tables unless absolutely necessary.
- Avoid generic ML explanations.
- Avoid repeating the same advice across sections.
- Do not include Markdown citations or source paths.

Quality bar: The final skill should clearly help a downstream solver outperform a generic strong Kaggle solution by:
- selecting a better task-specific modeling route,
- choosing stronger feasible models,
- aligning validation and metric handling,
- prioritizing high-ROI upgrades,
- avoiding attractive but wrong task-family transfers.

[Task-specific depth examples and final validation checklist omitted for brevity.]

After saving ./OUTPUTS/SKILL_<task>.md, print the output path, inferred task family/families, word count, and a brief validation note.

```

Table 18: Task-level skill synthesis prompt.

TASK-LEVEL SKILL EXAMPLE: MLSP-2013-BIRDS

Skill: mlsp-2013-birds

1. Task-Specific Reading

- This is weakly labeled bioacoustic multi-label classification. Each example is a ten-second mono WAV clip, and the target is the set of bird species present among 19 possible species.
- The useful prediction unit is the recording/species pair. Submit continuous probabilities or ranking scores for every species in every unlabeled test recording; do not collapse to a single species per clip.
- The metric is ROC AUC over binary labels and predictions. Treat it as a ranking metric:
 - thresholds are irrelevant for submission;
 - exact probability calibration matters less than ordering positives above negatives;
 - if the scoring implementation flattens recording/species pairs, cross-species score scale also matters, not just within-species ranking.
- The dataset is extremely small: 645 total recordings, with roughly half labeled and half unlabeled. A large deep model trained only on labels is high variance. The strongest strategy should combine stable feature-based models with one or more carefully regularized spectrogram models.
- The modality is audio, but the supplied preprocessing is unusually valuable:
 - raw WAVs allow log-mel, STFT, MFCC, chroma, spectral contrast, RMS/noise, and temporal summary features;
 - provided spectrogram and filtered spectrogram images allow image-style CNNs or deterministic image summaries;
 - provided segment features, segment rectangles, and histogram-of-segments features are direct multi-instance representations of bird-call-like regions and should not be ignored.
- Dominant score levers:
 - leakage-safe validation on training recordings with reliable multi-label stratification;
 - per-species OOF AUC plus flattened OOF AUC as the model-selection target;
 - strong one-vs-rest models on aggregated segment/audio features;
 - compact spectrogram CNN/SED predictions as a diverse second view;
 - OOF weight optimization/rank blending across model families;
 - class-wise monotonic transforms and clipping only when they improve OOF AUC.

2. Highest-Expected-Score Strategy

- Converge toward a hybrid ensemble with two complementary pillars:
 - Feature ensemble: one-vs-rest LightGBM/XGBoost/CatBoost, calibrated logistic/ridge, ExtraTrees/RandomForest, and possibly small MLPs over fixed recording-level features.
 - Spectrogram ensemble: small-to-medium CNN/SED models trained on log-mel or provided filtered spectrogram images with strong regularization, used for diversity rather than as the only signal.
- Make the feature ensemble the backbone of the solution. With only a few hundred labeled clips, handcrafted and supplied segment features are more statistically stable than fine-tuning large audio transformers.
- Build recording-level features from every available audio representation:
 - histogram-of-segments vector as a first-class feature block;
 - aggregate the 38-dimensional segment features per recording using count, mean, std, min, max, median, quantiles, skew-like summaries, and top-k/energy-like summaries when segment scores or rectangles allow them;
 - include segment count, missing-segment indicator, rectangle area/aspect/frequency/time-position aggregates, and proportions of segments in low/mid/high frequency bands;
 - compute raw WAV features: log-mel band statistics, MFCC mean/std/quantiles, spectral centroid/bandwidth/rolloff/flatness/contrast, zero-crossing rate, RMS/energy, high-frequency energy ratios, temporal energy quantiles, and simple noise/rain proxies;
 - compute image summaries from provided spectrograms/filtered spectrograms: global intensity quantiles, per-frequency and per-time projections, connected-component-like activity if simple, and contrast/noise-filter difference features.
- Train separate binary models for each species instead of forcing a multiclass objective. Multi-label positives can co-occur, and BCE/one-vs-rest objectives align with the label structure.
- For GBDT:
 - use regularization suitable for tiny data: shallow or moderate depth, subsampling, column sampling, large enough leaf sizes, and early stopping on folds;
 - train LightGBM on CPU; use XGBoost with `early\_stopping\_rounds` in the constructor if used;
 - train one model per species, with class weighting or sample weighting for rare species, but cap weights to avoid making a few positives dominate.
- For linear models:
 - use StandardScaler/RankGauss views and LogisticRegression or RidgeClassifier/SGD-style linear heads converted to scores;
 - these often produce robust rankings on small-N high-dimensional feature sets and add blend diversity.
- For spectrogram models:
 - prefer TimmSED-style CNN with attention pooling or a compact multi-label CNN over a huge foundation model;
 - use log-mel spectrograms from the 10-second clips, ideally 128 or 256 mel bins, hop length fine enough to preserve short bird calls;
 - train with BCEWithLogits or focal BCE, mixup with multi-label targets, light SpecAugment, and early stopping by OOF AUC;
 - if using provided filtered spectrogram BMPs, treat them as denoised image inputs and compare with raw log-mel, because stationary-noise suppression is likely useful in wind/rain clips.
- Blend by OOF predictions, not by standalone intuition:
 - optimize weights on identical folds using flattened AUC and mean per-class AUC;
 - for AUC, rank-transform predictions before blending can be strong, but keep a raw-probability blend candidate because cross-class scale may matter;
 - use per-class blend weights if OOF positives are sufficient, otherwise use global family weights to reduce overfitting.
- Do not depend on external datasets, species-range metadata, private labels, or non-local pretrained audio checkpoints as the default path. Use preinstalled or locally available timm/image weights only if the execution environment already provides them; the core plan must remain competitive from supplied data and installed libraries.

3. Strong First Implementation Plan

- Build one complete solution script around a serious OOF-driven hybrid feature ensemble.
- Parse labels into a binary target matrix of shape recordings x 19. Use only training-fold recordings for model fitting and unlabeled test recordings for final prediction.
- Create a reliable feature matrix before any deep learning:
 - load histogram-of-segments as direct numeric columns;
 - aggregate segment features per recording with count, mean, std, min, max, median, 10/25/75/90 percentiles, and missing indicators;
 - aggregate segment rectangles: number of segments, total/mean/max box area, aspect ratio stats, x/y center stats, width/height stats, and frequency-band occupancy;
 - compute WAV-level features from the 10-second waveform at native or consistent sample rate: MFCC, log-mel band means/stds/quantiles, spectral contrast, centroid, rolloff, bandwidth, flatness, RMS, ZCR, low/mid/high frequency energy ratios, and time-sliced energy summaries;
 - add summaries from supplied spectrogram and filtered spectrogram images if easy: intensity distribution, row/column projection quantiles, and filtered-minus-unfiltered contrast/noise features.
- Use multiple preprocessing views:

- raw numeric features with median imputation;
- standardized features for linear and MLP models;
- RankGauss or QuantileTransformer view for linear/NN diversity;
- optional PCA/SVD components on high-dimensional spectral blocks, concatenated with original features rather than replacing them.
- Validation:
 - use iterative multi-label stratification if implemented compactly; otherwise stratify by a compact label signature such as label count plus rarest/primary positive class, and verify every fold has positives for most species;
 - use 5 folds as the first reliable default. With very rare species, reduce fold count or force rare positives into train/validation coverage rather than creating folds with zero positives.
- First model set:
 - per-species LightGBM binary models on raw/engineered features with conservative regularization;
 - per-species XGBoost or CatBoost as diversity if time permits;
 - standardized LogisticRegression/Ridge/SGD one-vs-rest models for stable linear rankings;
 - ExtraTrees/RandomForest one-vs-rest for nonlinear diversity;
 - optional small MLP on RankGauss features with heavy dropout only if it is quick and OOF-tracked.
- Training target/loss:
 - binary targets per species;
 - use class weights or positive weights only where needed for rare species, capped and validated by AUC;
 - optimize model selection using OOF AUC, not logloss.
- Inference/postprocessing:
 - average fold predictions for each model family;
 - build OOF and test prediction matrices for each family;
 - blend raw predictions and rank-normalized predictions as separate candidates;
 - tune global and per-family weights on OOF flattened AUC, then confirm mean per-species AUC does not collapse for rare classes;
 - clip outputs to a small open interval only for numerical sanity; clipping should not change ranking materially.
- Add one compact spectrogram CNN only after the feature pipeline is stable:
 - train on log-mel or filtered spectrogram images at modest resolution;
 - use BCEwithLogits, mixup, light SpecAugment, and early stopping;
 - include its fold OOF/test predictions in the same blender, even if standalone AUC is weaker, because it may rank examples differently from tabular features.

4. High-ROI Upgrades Across Rounds

- [Later-round upgrade bullets omitted for brevity. The original skill continues with Round 2, Round 3, and late-round guidance on feature expansion, compact spectrogram models, TTA, OOF stacking, monotonic transforms, pseudo-labeling, and blend-weight search.]

5. Validation and Metric Optimization

- [Detailed validation and metric-optimization bullets omitted. The original skill specifies recording-level folds, multi-label stratification, flattened and per-species OOF AUC tracking, threshold-avoidance for AUC submissions, and OOF-based blend/transform selection.]

6. Model, Feature, and Preprocessing Priorities

- [Detailed priority lists omitted. The original skill enumerates feature blocks, model families, spectrogram preprocessing priorities, multi-label loss choices, and ensembling rules.]

7. Avoid or Delay

- [Avoid/delay list omitted for brevity. The original skill lists metric-mismatched single-label modeling, recording-level leakage, over-aggressive pseudo-labeling, unsafe external dependencies, and spectrogram augmentations that violate time/frequency semantics.]

Table 19: Task-level skill example.

FAILURE PREVENTION SKILL EXAMPLE

ML Failure Prevention Skill: Plan-Code Version

Purpose

Read this skill before planning and while writing a one-shot ML solution script.

Each round follows this workflow:

1. Use this skill before coding to make an explicit contract plan.
2. Write one complete ML solution script.
3. Run the script in the sandbox.

The script is an end-to-end Kaggle-like solution: data loading, preprocessing, validation, modeling, inference, and writing the required submission file.

The purpose is to prevent brittle assumptions that break otherwise strong, high-scoring approaches, while preserving ambition in modeling, feature engineering, ensembling, deep learning, pretrained encoders, and modality-specific methods.

This skill is not a request to expose hidden chain-of-thought. The pre-code plan should be a concise, visible contract checklist: what the code must preserve, align, validate, and output.

Core Principle

Build ambitious but contract-aware solutions.

Code design MUST be driven by:

- observed data schema;
- fixed-sandbox API compatibility;
- feature/model compatibility;
- valid split assumptions;
- explicit array/tensor/container interfaces;
- modality-aware preprocessing;
- output-schema construction by design.

Do not weaken modeling just to avoid contract work. Strong models, deep learning, ensembling, pretrained encoders, rich feature engineering, and modality-specific methods are encouraged when appropriate, but their data, target, split, tensor, and output contracts MUST be satisfied.

Workflow Requirements

A. Before Coding: Explicit Contract Plan

Before writing code, produce a concise plan that answers the relevant contracts below.

The plan MUST identify:

- available files and how schema will be derived from loaded data;
- target, identity, feature, metadata, auxiliary, join-key, and output roles;
- model family, feature representation, target encoding, objective/loss/metric, and prediction representation;
- train/test feature alignment strategy;
- validation split strategy and why it is valid for the target/data structure;
- dataframe/numpy/sparse/tensor/list/dict boundaries, including shape, dtype, device, and row order;
- modality-specific irregularities and how preprocessing will handle them;
- output schema, row identity, class/order mapping, and prediction-to-submission conversion.

The plan MUST NOT rely on remembered Kaggle patterns, common column names, common APIs, common tensor shapes, common label spaces, or common output formats.

B. During Coding: Make Contracts Concrete

While writing code, relevant contracts MUST be implemented through code structure, explicit transformations, checks, alignment logic, or safe fallbacks.

A runtime guard must not turn a recoverable contract violation into a full round failure. If the script can choose a semantically valid fallback, it must do so and log it. Only raise when continuing would create invalid labels, misaligned rows, or semantically wrong output.

Do not merely mention safeguards in comments. If a contract matters, the script must preserve it, check it, or make violation impossible by construction.

Failure Prevention Contracts

1. Environment and API Contract

The runtime sandbox dependencies are fixed. Do not suggest upgrading, installing, or changing them. Write code that works with the available dependency set.

[Detailed requirements and checklist omitted for brevity.]

2. Data Schema Contract

Derive schema from loaded files, not from memory, prompt patterns, or generic task templates.

Treat columns, indexes, join keys, generated features, row identity, table shape, and train/test feature alignment as design contracts.

[Detailed requirements and checklist omitted for brevity.]

3. Feature, Target, and Model Compatibility Contract

Choose the model, feature representation, target encoding, objective, loss, metric, sparsity, missingness, dtype, and value domain as one compatible system.

Strong models only help if the produced data satisfies their contracts.

[Detailed requirements and checklist omitted for brevity.]

4. Validation Split Contract

Design validation from the target distribution and data-generating structure.

Do not blindly apply stratification, folds, random splits, or grouped splits because they are common templates.

[Detailed requirements and checklist omitted for brevity.]

5. Array, Tensor, and Batch Interface Contract

Every conversion boundary needs explicit container, shape, rank, dtype, device, and row-order semantics.

This includes transitions among dataframes, numpy arrays, sparse matrices, tensors, Python lists, dictionaries, datasets, dataloaders, prediction arrays, and model-specific batches.

[Detailed requirements and checklist omitted for brevity.]

6. Modality Edge-Case Contract

Generic recipes break on modality irregularities. Modality-aware preprocessing should enable stronger models by handling irregular samples before they reach the model.

[Detailed requirements and checklist omitted for brevity.]

7. Output Schema Contract

The output format is a design constraint from the start, not an afterthought.

[Detailed requirements and checklist omitted for brevity.]

Compact Mandatory Heuristics

- MUST NOT assume schema, API, dtype, shape, label space, row order, class order, or output format from memory.
- MUST load data first, derive schema, and make target, identity, feature, and output roles explicit.
- MUST choose model and representation as one contract: features, target, objective, metric, and output must agree.
- MUST keep row identity alive through preprocessing, splitting, batching, prediction, ensembling, and output writing.
- MUST construct train/test features symmetrically and explicitly align feature spaces.
- MUST treat categorical mappings, class order, token labels, thresholds, and inverse transforms as persistent artifacts.
- MUST choose validation based on target support, grouping, multilabel structure, and time/order constraints.
- MUST define every dataframe/numpy/sparse/tensor/list/dict boundary by container, shape, dtype, device, and row order.
- MUST design collation, padding, resizing, masking, and batching before using variable-size or structured samples.
- MUST use advanced libraries and modality-specific models when valuable, but avoid fragile version-specific APIs.
- MUST build output construction around the required external contract before training.
- MUST prefer robust ambitious plumbing over simpler modeling chosen only to avoid implementation risk.

What This Skill Is Not

- Not a traceback cookbook or a list of post-failure fixes.
- Not a reason to avoid ambitious models, deep learning, ensembling, feature engineering, or modality-specific methods.
- Not a request to upgrade, install, or change sandbox dependencies.
- Not post-hoc debugging advice or merely a final submission-only check.
- Not benchmark-specific and not tied to particular tasks, datasets, columns, paths, labels, error messages, or leaderboard quirks.
- Not a resource-conservatism rule derived from short-run failures; manage resources as part of sound modeling, not as a reason to default to weak baselines.

Table 20: Failure prevention skill example.