PRISM-DS: Closing the Task-Profile-to-Skill Loop for Data-Science Agents

Anonymous ACL submission

Abstract

Large language models (LLMs) have enabled automating end-to-end data-science pipelines, yet robust solutions still require aligning modeling choices with the data, the evaluation metrics, and task-specific challenges. Existing agents often lack explicit diagnostic mechanisms; instead, they reply on parametric memory or external resources that are weakly tied to the problem. This often results in generic or misaligned designs. To bridge this gap, we propose PRISM-DS, a framework that closes the loop between profiling and skill retrieval. PRISM-DS employs a profiler to extract structured task characteristics, which serve as queries for an LLM retrieval agent. PRISM-DS then searches a taxonomy-aligned knowledge base of 4,421 curated skills distilled from 1,504 medal-winning notebooks, with optional web tools. These retrieved skills guide initial solution generation. During block refinement, the retrieval is performed again by combining the task characteristics with the stage specific pipeline to obtain stage-targeted skills. To ensure reliable candidate selection, we further integrate fixed-split validation and submission verifier. PRISM-DS consistently outperforms strong baselines across three benchmarks: it achieves 64.01% RPG on DS-Bench (GPT-5), outperforming the best baseline by 9 percentage points; it reaches 48.38 on DareBench modeling tasks; it secures medals on 52.94% of MLE-Bench-Lite tasks, versus 35.29% for prior leading methods.

1 Introduction

Data science tasks span a large space of methods, and strong pipelines depend on matching those choices to the data and the evaluation objective. Human experts typically diagnose the task first, then gather targeted skills before coding. LLM-based agents (Guo et al., 2024; Jiang et al., 2025; Nam et al., 2025; Liu et al., 2025; Du et al., 2025; Li et al., 2026) can already generate and refine executable solutions on benchmarks, yet most still choose methods from broad parametric memory without an explicit notion of what makes the current task hard, which yields generic or misaligned designs.

We treat structured task diagnosis as a necessity: modality, task type, target metric, and evidenced challenges must be identified before modeling choices are justified, yet prior systems largely skip this step. Agents such as AIDE (Jiang et al., 2025) and ML-Master (Liu et al., 2025) rely on frozen parametric memory (Nam et al., 2025); DS-Agent (Guo et al., 2024) retrieves fixed cases by surface similarity; MLE-STAR (Nam et al., 2025) searches the web without a rigorous data-side analysis; and recent search- or forum-augmented variants (Du et al., 2025; Zhu et al., 2026; Li et al., 2026) still do not turn diagnosed challenges into a precise retrieval specification. As a result, even when relevant external skills exist, it is hard to tell what is relevant, and retrieval drifts toward the wrong subspace.

Diagnosis must be paired with high-quality, task-targeted skills. Parametric weights go stale; previously curated case banks are unstructured and narrow (Guo et al., 2024); and web results are noisy and weakly constrained (Nam et al., 2025). What is missing is a curated resource organized along task-relevant axes, together with retrieval that can exploit a fine-grained diagnostic signal instead of a fixed query pattern. In realistic user tasks, however, exact matching skills are often unavailable, so the system must use diagnosed task characteristics to retrieve the most relevant skills rather than look for a perfect match.

To bridge this gap, we present **P**rofile-guided **R**etrieval of **I**mplementable **S**kills for **M**odeling in **D**ata **S**cience (PRISM-DS), built around a task-profile-to-skill loop. Specifically, a profiler first summarizes each task into structured task char-

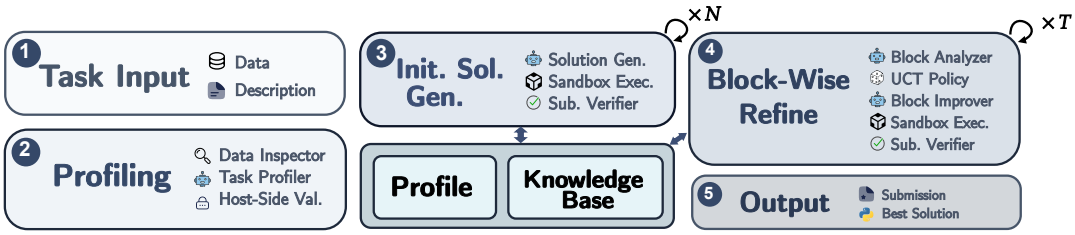


Figure 1: Overall workflow of PRISM-DS. The system first profiles the task into a structured report, retrieves curated skills from the KB, generates initial candidates under a shared retrieval result, and then performs stage-aware refinement under a fixed budget. Figures 2 provides detailed views of the profile-guided retrieval process.

acteristics, including modality, target metric, and evidenced challenges grounded in the input data. These task characteristics then serve as the retrieval query. Conditioned on this query, an LLM retrieval agent actively explores a organized and structured skill knowledge base (KB) of 4,421 curated skills distilled from 1,504 medal-winning notebooks (with optional web tools), deciding which categories, stages, and skills to inspect through tool calls rather than following a fixed retrieval recipe. The retrieved skills guide initial solution generation. Moreover, the same loop supports block-level refinement by re-running retrieval with the task characteristics and the selected pipeline stage, enabling more targeted updates. We further pair these core components with additional optimisation modules that improve candidate selection and refinement reliability. On DSBench (Jing et al., 2024), PRISM-DS reaches 64.0% RPG with GPT-5 (9 percentage points over the best baseline at convergence, with monotonic gains across steps); on DareBench (Shu et al., 2026) modeling tasks, it reaches 48.38, outperforming all baselines; on MLE-Bench-Lite (Chan et al., 2024), it earns medals on 53% of tasks with a 100% valid-submission rate. A controlled ablation shows that the profiler–KB retrieval loop alone improves RPG by 7.64 percentage points over an All-Off agent, indicating that the core loop rather than supporting checks drives most of the gain.

Our contributions are:

- **We build and will publicly release a curated, four-level, stage-tagged KB of implementable skills** that enables challenge-aware and stage-aware retrieval at scale.
- We are the **first** to introduce a task-profile-to-skill loop that profiles each task into structured task characteristics and uses them to drive agentic retrieval of curated skills.
- We pair the loop with fixed-split validation and related checks, achieving strong results on DSBench, DareBench modeling tasks, and MLE-Bench-Lite under CPU-only budgets across proprietary and open-source LLM backbones.

2 Related Work

LLM-based Agents. Large language models have been used as autonomous agents that reason, plan, and call external tools (Yao et al., 2023; Schick et al., 2023), with applications ranging from software engineering (Jimenez et al., 2024) to game-playing (Wang et al., 2023). Data science is a natural next target, but it poses additional challenges: agents must handle diverse data modalities, select appropriate modeling strategies, and produce executable end-to-end pipelines.

Data Science and ML Engineering Agents. DS-Agent (Guo et al., 2024) retrieves strategies from a Kaggle case bank, but manual curation limits its coverage and scalability. AIDE (Jiang et al., 2025) and ML-Master (Liu et al., 2025) organize candidates via tree search but rely solely on parametric knowledge, risking outdated choices (Nam et al., 2025). MLE-STAR (Nam et al., 2025) incorporates web search and ablation-guided refinement, yet lacks upfront task profiling to condition retrieval on specific challenges (e.g., class imbalance). Unlike classical AutoML (Kotthoff et al., 2017; Olson et al., 2016; Erickson et al., 2020) which searches predefined spaces constrained by domain expertise, LLM agents operate flexibly in code space, necessitating better mechanisms to steer exploration toward task-appropriate techniques.

Data Science Agent Benchmarks. Early coding benchmarks test isolated functions (Chen et al., 2021; Austin et al., 2021) or repository-level bug fixes (Jimenez et al., 2024), while DS-1000 (Lai

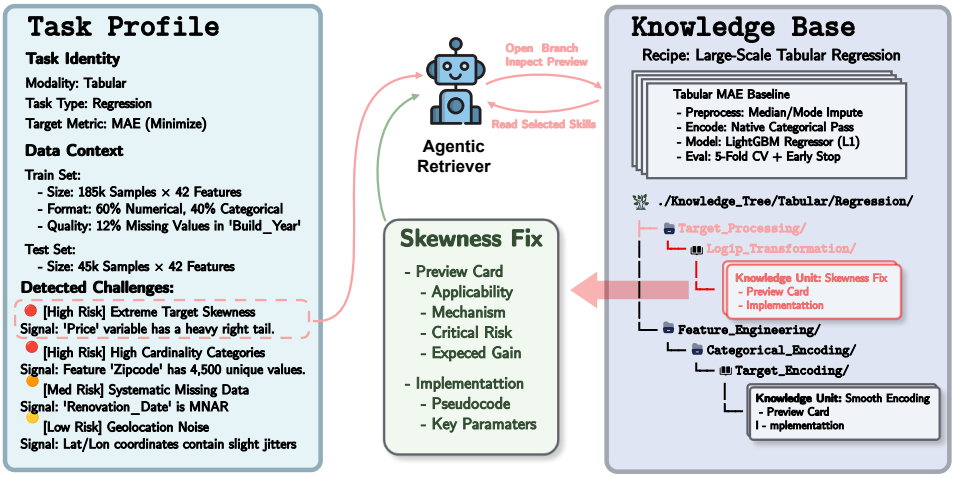


Figure 2: Profile-guided agentic retrieval. The task profile provides the information used to construct the retrieval query, and the retrieval agent actively navigates the curated KB to assemble task-relevant skills.

et al., 2022) evaluates library API usage without requiring full pipelines. DSBench (Jing et al., 2024) and MLE-bench (Chan et al., 2024) use Kaggle-style competition tasks that demand the complete workflow from data loading to submission. Other benchmarks target specific aspects such as financial analytics (Egg et al., 2025), structured code evaluation (Zhang et al., 2025), private-data scenarios (Kadiyala et al., 2025), and process-aware instruction fidelity (Shu et al., 2026). We evaluate on DSBench, Darebench and MLE-bench as they best cover end-to-end data science pipelines with quantitative scoring.

3 PRISM-DS

We now describe how PRISM-DS implements the task-profile-to-skill loop introduced in the introduction. We will begin by formalizing the task setting for the data science agent problems. We then present the overall pipeline of PRISM-DS. We next detail how the core components realizes such task-profile-to-skill loop, followed by additional supporting reliability modules that further improve the agents' performance. Figure 1 summarizes the overall workflow; Figures 2 and 3 zoom in on retrieval and KB construction; Algorithm 1 gives the end-to-end procedure.

3.1 Task Formulation

Given a task description c, an original labeled training set $\mathcal{D}_{\text{train}}^{\text{orig}} = \{(x_i, y_i)\}_{i=1}^{n}$, and an unlabeled test set $\mathcal{D}_{\text{test}} = \{x_j\}_{j=1}^{m}$ and the corresponding labels $Y_{\text{test}} = \{y_j\}_{j=1}^{m}$ that are unavailable to the model. The goal is to construct a prediction pipeline that produces the labels $\hat{Y}_{\text{test}}$ for the unlabeled test set. The quality of the pipeline is measured by a task-specific metric $m(Y_{\text{test}}, \hat{Y}_{\text{test}})$. During execution, PRISM-DS further splits $\mathcal{D}_{\text{train}}^{\text{orig}}$ into an internal training subset $\mathcal{D}_{\text{train}}$ and a validation subset $\mathcal{D}_{\text{valid}}$ for reliable model selection.

3.2 Overall workflow of PRISM-DS

PRISM-DS runs in three stages: profiling, initialization, and refinement. **In the profiling stage,** the profiler agent inspects the task description and the available data, and summarizes them into a structured report P that captures the information later used to guide both retrieval and generation.

In the initialization stage, the retrieval agent runs once on P to obtain a shared skill bundle b_0. The generator then produces candidates sequentially using this same bundle. Each candidate receives solutions and validation scores of previous candidates to encourage generation diversity, with no additional retrieval performed for cost saving. In contrast, during the subsequent refinement iterations, retrieval is rerun for the pipeline stage selected at each iteration.

In the refinement stage, revision will be carried out on best solution from initialization stage (e.g. feature_engineering, model_selection). PRISM-DS first parses this solution into pipeline stages and iteratively improves one stage at a time. At each refinement step, the Upper Confidence bounds applied to Trees (UCT) criterion (Appendix C) selects a stage to revise, the refinement agent retrieves stage-specific skills for that stage, and the agent proposes a targeted update for execution only when

the change appears promising by evaluator agent.

3.3 Core Components

The workflow above is supported by three core components. Together, they implement the task-profile-to-skill loop that underlies both initialization and refinement: (1) a carefully curated, taxonomy-organized knowledge base of skills, (2) a structured profiling module that extracts task characteristics, and (3) an agentic retrieval mechanism that connects the two. We introduce these components in this order.



Figure 3: Overview of KB curation. Medal-winning notebooks are segmented into skill units, organised into a taxonomy, and stored with structured metadata for later retrieval.

Curated taxonomy-organized KB of skills. Shown in Figures 3, we curate a knowledge base $\mathcal{B}$ from gold-medal Kaggle notebooks, indexed on the same axes as the profiler. The skill content originates from human-written notebooks: rather than generating implementations from scratch, an LLM librarian segments and organizes the source material into skill units, assigns each to a leaf of a branching four-level tree Modality $\rightarrow$ Task Type $\rightarrow$ Challenge $\rightarrow$ Solution Family (e.g. Time Series $\rightarrow$ Classification $\rightarrow$ Feature Engineering $\rightarrow$ Aggregation), and annotates units with preview cards, implementation blocks, pipeline-stage tags, and applicability conditions. Extracted units are relevance-filtered before indexing, and their conditions serve as checkable predicates against the task profile P. **At retrieval time, skills whose source competition matches the evaluation task are excluded to prevent leakage.** The KB contains 4,421 skills from 1,504 medal-winning notebooks (5 modalities, 13 task types, 21 challenge categories, 30 solution families, 13 code stages, 27 full-pipeline recipes); see Appendix J.

Structured profiling and task characteristics. Before any solution is generated, PRISM-DS profiles the task by exploring $\mathcal{D}_{\text{train}}^{\text{orig}}$, $\mathcal{D}_{\text{test}}$, and the task description c, summarizing them into a structured report P that records the data setting, evaluation metric, and evidenced challenges. A data inspector first collects task evidence such as available files, row counts, column types, missing-value statistics (including which columns contain missing values and at what rates), per-column value distributions, and modality-specific artifacts. Given this overview and the task description, an LLM infers modality, task type, evaluation metric m and its direction, and a list of likely challenges (e.g., class imbalance, missing values, or temporal structure), each paired with short evidence. The report P, which connects the profiling stage to the downstream initialization and refinement stages, is then used both to construct retrieval queries and to ground generation prompts, keeping retrieval and coding aligned with the same diagnosis of the task.

Agentic retrieval of curated skills. Prior work often retrieves with a fixed procedure (e.g., top-k case matching or a single web query). In contrast, PRISM-DS, shown runs a retrieval agent with function-calling tools that reads the task profile P and actively decides which axes of knowledge base $\mathcal{B}$ to open, which filters to apply, and which skills to inspect in full. Available tools include browsing taxonomy branches, fetching by challenge or code stage, semantic search over skills, expanding implementation text, and optional web queries. Each candidate skill is checked against profile-side applicability conditions and then passed through an LLM relevance filter, so only task-appropriate skills reach code generation. If no KB branch or skill satisfies these conditions, the retriever falls back to open-web search and applies the same relevance filter to the retrieved guidance. For initialization, the retriever returns a shared skill bundle b_0 that is used by all initial candidates. During refinement, retrieval is run again with the selected pipeline stage as additional context, producing a stage-specific bundle b_t. Since the trajectory over tools is not hard-coded, the design is agentic. The agent can deepen search when the profile is ambiguous, skip branches ruled out by evidence, or pivot after inspecting preview cards. With these three components in place, PRISM-DS can retrieve targeted skills before generation and retrieve them later during stage-level refinement.

3.4 Additional Supporting Reliability Modules

After grounding the system with the three core components above, we add supporting reliability modules that stabilize scoring, enforce submission correctness, and control execution.

Algorithm 1 PRISM-DS Pipeline

Input: task description c; labeled data $\mathcal{D}_{\text{train}}^{\text{orig}}$; unlabeled data $\mathcal{D}_{\text{test}}$; skill KB $\mathcal{B}$
Output: best solution $solution_{\text{best}}$ and predictions $\hat{Y}_{\text{test,best}}$

$P \leftarrow \textsc{Profiler}(c,\ \mathcal{D}_{\text{train}}^{\text{orig}})$
Split $\mathcal{D}_{\text{train}}^{\text{orig}}$ once into fixed $\mathcal{D}_{\text{train}}$ and $\mathcal{D}_{\text{val}}$
$(solution_{\text{best}}, score_{\text{best}}, \hat{Y}_{\text{test,best}}) \leftarrow (\emptyset, -\infty, \emptyset)$
$b_0 \leftarrow \textsc{Retriever}(P, \mathcal{B})$ ▷ profile-guided initialization skills
Initialization:
for $n = 1$ **to** N **do**
  $solution_n \leftarrow \textsc{InitialGenerator}(P, b_0, \{(solution_i, score_i)\}_{i<n})$
  $(\hat{Y}_{\text{test},n}, \hat{Y}_{\text{val},n}) \leftarrow \textsc{ExecDebugger}(solution_n, \mathcal{D}_{\text{train}}, \mathcal{D}_{\text{val}})$; $score_n \leftarrow m(\hat{Y}_{\text{val},n}, Y_{\text{val}})$
  if $score_n > score_{\text{best}}$ **then**
    $(solution_{\text{best}}, score_{\text{best}}, \hat{Y}_{\text{test,best}}) \leftarrow (solution_n, score_n, \hat{Y}_{\text{test},n})$
  end if
end for
Parse $solution_{\text{best}}$ into blocks $\{q_1, \ldots, q_M\}$
Block-wise refinement:
for $t = 1$ **to** T **do**
  $i \leftarrow \arg\max_j \text{UCT}(q_j)$; $b_t \leftarrow \textsc{Retriever}(P, \mathcal{B}, q_i)$ ▷ stage-conditioned skills
  $(q_i', summary) \leftarrow \textsc{Refiner}(solution_{\text{best}}, q_i, b_t, score_{\text{best}})$
  if $\textsc{Evaluator}(q_i', summary)$ **then**
    $solution_t \leftarrow \textsc{Replace}(solution_{\text{best}}, q_i, q_i')$
    $(\hat{Y}_{\text{test},t}, \hat{Y}_{\text{val},t}) \leftarrow \textsc{ExecDebugger}(solution_t, \mathcal{D}_{\text{train}}, \mathcal{D}_{\text{val}})$; $score_t \leftarrow m(\hat{Y}_{\text{val},t}, Y_{\text{val}})$
    if $score_t > score_{\text{best}}$ **then**
      $(solution_{\text{best}}, score_{\text{best}}, \hat{Y}_{\text{test,best}}) \leftarrow (solution_t, score_t, \hat{Y}_{\text{test},t})$
    end if
  end if
  Update block-i UCT statistics and decay the exploration constant
end for
return $solution_{\text{best}}, \hat{Y}_{\text{test,best}}$

Fixed-split validation After profiling and before initialization, we split $\mathcal{D}_{\text{train}}^{\text{orig}}$ once into $\mathcal{D}_{\text{train}}$ and $\mathcal{D}_{\text{val}}$, and fix them for all later candidates. This removes the need for sandboxed code to create its own split and ensures that every candidate is evaluated on the same held-out data. Otherwise, different candidates could be scored on different validation sets, so higher validation would not reliably mean a better solution.

Submission verifier Some generated solutions run without execution errors but still produce invalid submissions. We therefore add a rule-based verifier that checks row count, identifier alignment, and output structure against the task specification. If execution succeeds but verification fails, we treat the result as a bug and send it back to debugging rather than counting it as a valid candidate.

Evaluation agent Before execution, we send each refinement candidate and a short summary of its proposed changes to an evaluation agent, which predicts whether it is likely to improve over the current best solution. If the predicted gain is negative or unconvincing, we skip execution for that refinement step; otherwise, the candidate is executed in the sandbox and scored on the shared fixed-split validation split. If execution fails, or if the submission violates the required format, an LLM debugger proposes fixes up to a retry limit.

Comparison with existing frameworks Table 1 separates PRISM-DS from prior agents along the same split as this section. On the **core** side, existing systems generally lack a profiler that compiles metric and challenges into task characteristics, a taxonomy-organised KB aligned with those characteristics, or retrieval that is both diagnosis-conditioned and actively planned by an LLM agent rather than fixed upfront. On the **additional optimisation** side, PRISM-DS combines fixed-split validation, submission verification, and an evaluation agent to provide trustworthy rewards for candidate selection and refinement acceptance; the baselines omit one or more of these checks.

4 Experiments

We evaluate whether the task-profile-to-skill loop improves end-to-end competition-style data science: task characteristics used as retrieval queries, together with agentic retrieval of curated skills,

Table 1: Component comparison across data-science agent frameworks. Rows under **Core** align with the task-profile-to-skill loop; rows under **Supporting modules** are orthogonal engineering choices. ✓ = present, ✗ = absent, ∼ = partial.

Component	DS-Agent	ML-Master	MLE-STAR	PRISM-DS (Ours)
Core (diagnosis + targeted knowledge)				
Task profiler / task characteristics	✗	✗	✗	✓
Structured KB + taxonomy	∼	✗	✗	✓
Agentic skill retrieval	✗	✗	✗	✓
Supporting modules				
Online search	✗	✗	✓	✓
Multi-candidate init. soln.	✗	✓	✓	✓
Block-level refinement (UCT)	✗	✓	✓	✓
Host-side validation	✗	✗	✗	✓
Submission verifier	✗	✗	✓	✓
Evaluation agent (LLM)	✗	✗	✗	✓

should raise initial solution quality and support stable refinement.

4.1 Experimental Setup

Baselines. We compare PRISM-DS against three state-of-the-art data-science agents. **DS-Agent** (Guo et al., 2024) retrieves human-written case studies from a curated case bank and conditions code generation on the retrieved strategies. **ML-Master** (Liu et al., 2025) organizes solutions in a search tree and applies block-level edits, but does not use an external knowledge base. **MLE-STAR** (Nam et al., 2025) combines web search for up-to-date modeling ideas with ablation-guided block refinement, without a structured task-profiling stage before retrieval.

Benchmarks and metrics. We evaluate on three benchmarks. (1) **DSBench** (Jing et al., 2024) contains 74 competition-style tasks spanning tabular, time-series, and mixed modalities. We follow the benchmark and report Relative Performance Gap (RPG, %), defined as $\frac{1}{N}\sum_{i=1}^{N}\max((p_i-q_i)/(g_i-q_i),0)$, where p_i is the agent score, g_i the expert score, and q_i the benchmark baseline on task i. (2) **DareBench** (Shu et al., 2026): we report the score on its modeling subset. (3) **MLE-Bench-Lite**: we start from 22 MLE-Bench tasks (Chan et al., 2024), remove 5 with data larger than 1 GB, and evaluate on the remaining 17 using valid-submission and medal rates. Best-checkpoint / solution selection rules are in Appendix D.

Compute resources and budgets. All experiments are conducted on **CPU-only machines**. To ensure a fair comparison, each framework is strictly allocated **10 solution steps** (excluding up to 3 debug retries per step). Online searches are powered by Serper (Serper.dev, 2026). For PRISM-DS, this comprises 5 initial candidates (N=5) and 5 refinement iterations (T=5). All methods within each comparison use the same LLM backbone; in particular, **every MLE-Bench-Lite method uses GPT-5 with the same step and retry budgets**. For repeated experiments, we report mean performance and standard deviation.

4.2 Main Results

Results on DSBench. Figure 4 plots RPG across refinement steps; PRISM-DS is consistently the best under both LLMs. With GPT-5, RPG rises from 58.06% to 64.01% at step 9 (vs. DS-Agent 48.57% → 54.99%), maintaining the lowest variance at convergence (std 0.46). With Sonnet 4.5, RPG improves from 57.82% to 59.58% (vs. ML-Master 56.14%). Sonnet starts closer to the ceiling, whereas GPT-5 exhibits broader exploration; baselines share this pattern, reflecting inherent LLM behaviors. While baselines plateau early (ML-Master peaks at step 2; MLE-STAR stagnates at 53–55%), we attribute our sustained gains to profile-guided retrieval and stage-conditioned UCT. For efficiency, PRISM-DS takes 79.35 minutes per task, faster than ML-Master and DS-Agent, as shown in Table 3. Under matched wall-clock cutoffs, PRISM-DS reaches 54.5% within one hour, exceeding every baseline's four-hour result (Table 14); evaluation details are in Appendix H.

Results on MLE-Bench-Lite. Table 2 summarizes results on our filtered MLE-Bench-Lite subset, with all four methods using the same GPT-5 backbone and matched budgets. PRISM-DS is the only framework with a 100% valid-submission

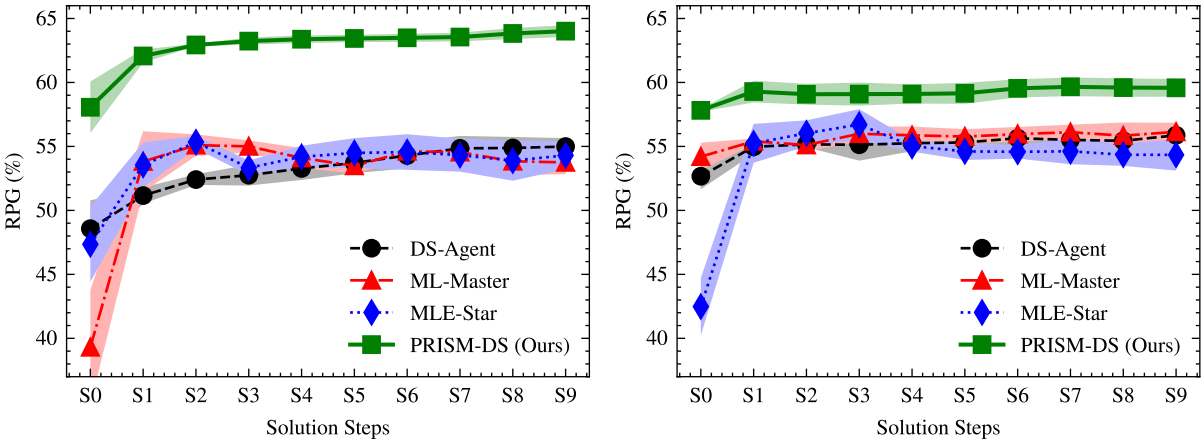


Figure 4: Step-wise RPG curves on DSBench (Jing et al., 2024). Left: GPT-5. Right: Claude Sonnet 4.5. X-axis: solution step (0–9); Y-axis: average RPG (%). Exact numbers could be found in Appendix F.

Table 2: Results on sampled MLE-Bench-Lite (Chan et al., 2024) (17 tasks, data size no larger than 1 GB). Every method uses GPT-5 and identical step/retry budgets. Metrics are percentages.

Framework	Valid Submission	Above Median	Bronze	Silver	Gold	Any Medal
DS-Agent	76.47$_{\pm 9.61}$	29.41$_{\pm 4.88}$	0.00$_{\pm 0.00}$	5.88$_{\pm 0.00}$	9.80$_{\pm 2.77}$	15.69$_{\pm 2.77}$
ML-Master	94.12$_{\pm 4.88}$	56.86$_{\pm 7.34}$	17.65$_{\pm 0.00}$	13.73$_{\pm 2.77}$	3.92$_{\pm 2.77}$	35.29$_{\pm 4.88}$
MLE-STAR	94.12$_{\pm 0.00}$	29.41$_{\pm 0.00}$	0.00$_{\pm 0.00}$	11.76$_{\pm 4.88}$	5.88$_{\pm 4.88}$	17.65$_{\pm 4.88}$
PRISM-DS (Ours)	**100.00**$_{\pm 0.00}$	**76.47**$_{\pm 9.61}$	**21.57**$_{\pm 2.77}$	**15.69**$_{\pm 2.77}$	**17.64**$_{\pm 4.80}$	**52.94**$_{\pm 0.00}$

rate, compared with 76% to 94% for the baselines, which is consistent with the effect of fixed-split validation and submission verification. It earns medals on 52.94% of tasks, compared with 35.29% for the next-best system (ML-Master), and also achieves the best gold-medal rate (17.64%) and Above Median rate (76.47%). These results confirm that the advantage of PRISM-DS transfers beyond DSBench.

Table 4: DareBench (Shu et al., 2026) modeling-task results with GPT-5.

Framework	Score
DS-Agent	41.17$_{\pm 2.17}$
ML-Master	42.23$_{\pm 2.68}$
MLE-STAR	44.24$_{\pm 2.09}$
PRISM-DS	**48.38**$_{\pm 1.85}$

Results on DareBench. Table 4 shows the same pattern on DareBench modeling tasks: PRISM-DS reaches 48.38, outperforming MLE-STAR (44.24), ML-Master (42.23), and DS-Agent (41.17). This suggests that the gains from profile-guided skill retrieval are not specific to DSBench.

Table 5: DSBench RPG (%) at selected refinement steps with Qwen3-32B.

Framework	S1	S4	S9
DS-Agent	45.73	46.74	48.90
ML-Master	47.84	53.19	52.29
MLE-STAR	44.35	50.81	45.57
PRISM-DS	**50.75**	**54.73**	**55.71**

Open-source backbone. We additionally compare all four frameworks using Qwen3-32B (Qwen Team, 2025) on DSBench. Table 5 shows that PRISM-DS reaches 55.71% RPG at the final step, outperforming the strongest baseline, ML-Master (52.29%), by 3.42 points. This consistent advantage indicates that the benefit is not restricted to frontier proprietary backbones; the complete trajectory is in Appendix G.

4.3 Ablation Study

To quantify the contribution of each major component, we conduct ablations on DSBench using GPT-5 and report RPG@10. We isolate the core loop and each reliability module from an All-Off configuration, remove individual modules from the full system, and replace the curated KB with alternative skill sources.

Table 6: Bottom-up isolation on DSBench with GPT-5 (RPG@10, %).

Configuration	RPG@10
All Off	54.84$_{\pm 0.88}$
Evaluator only	55.37$_{\pm 0.85}$
Fixed validation only	57.43$_{\pm 1.00}$
Verifier only	57.87$_{\pm 0.74}$
Core loop only	**62.48**$_{\pm 0.43}$
All On	**64.01**$_{\pm 0.46}$

Table 7: Skill-source ablation on DSBench with GPT-5 (RPG@10, %). All rows retain the full profiling and generation pipeline; only the knowledge source and its retrieval mechanism are replaced.

Skill source	RPG@10
DS-Agent skills + embedding retrieval	62.27$_{\pm 0.86}$
Online search	62.69$_{\pm 0.98}$
Our curated KB	**64.01**$_{\pm 0.46}$

Bottom-up isolation. Starting from a vanilla All-Off configuration, we enable each reliability module alone or enable only the profiler–KB retrieval loop. The core loop raises RPG@10 from 54.84% to 62.48% (+7.64 points), whereas individual reliability modules reach 55.37–57.87%. Adding all modules on top of the core loop yields 64.01%, a further 1.53-point gain. Thus, reliability modules help, but the isolated core loop accounts for the primary improvement.

Skill source ablation. We also compare different skill sources under the same DSBench setup and GPT-5 backend; implementation details (embedding model, corpus, and online-search settings) are in Appendix E. Replacing our curated KB with DS-Agent-style case text plus dense retrieval reduces RPG@10 to 62.27%, and replacing it with online search only yields 62.69%. Thus, retrieval quality depends not only on the retrieval procedure but also on the structure and curation of the underlying skill source.

4.4 Detailed Analysis

Case study: `liverpool-ion-switching` (DS-Bench). Moving beyond the ablations above, we illustrate how profiling and KB retrieval shape the pipeline using the `liverpool-ion-switching` task (Jing et al., 2024) (macro-F1 classification of ion channel states). Final solution codes for all frameworks are archived in Appendix I. Here, measurements arrive in 50-second batches; smoothing across batch boundaries can leak information and hurt rare classes. **ML-Master** trains a 1D CNN reaching 40.9% RPG; its rolling statistics ignore batch masks, allowing windows to span boundaries. **MLE-STAR** reaches 34.3% RPG with CatBoost; it uses batch-grouped features but lacks per-batch normalization.

Conversely, PRISM-DS achieves 91.8% RPG (macro-F1 0.922 after refinement). The profiler identifies the multi-class time-series structure, class overlap, and batch-leakage risk, guiding KB retrieval toward per-batch normalization, within-batch rolling statistics, and class-reweighted gradient boosting. The generator then z-score normalizes within batches, adds lag features safely, and fits a LightGBM model with light temporal smoothing. The gain therefore comes from a sequence of compatible decisions rather than a single model choice, each feature and validation choice respects the same batch constraint. The retrieved skills also expose implementable operations, so refinement can adjust feature windows and model weights without switching to an incompatible pipeline. Without profiling and structured KB retrieval, agents rarely surface this exact combination under CPU limits.

Table 3: Average per-task runtime and LLM token usage on DSBench with GPT-5. Runtime is reported in minutes per task.

Framework	Avg time / task	Avg tokens / task
PRISM-DS	79.35	358,420
DS-Agent	318.01	234,275
ML-Master	154.46	322,870
MLE-STAR	59.84	331,615

5 Conclusion

We presented PRISM-DS, a data-science agent built around a task-profile-to-skill loop. Across DSBench, DareBench modeling tasks, and filtered MLE-Bench-Lite, PRISM-DS consistently outperforms prior agents; ablations show that profiling and curated skill retrieval drive the gains. Results with Qwen3-32B further indicate that the benefit of structured orchestration transfers across backbones.

For anonymous review, we provide the curated taxonomy, skill metadata, and KB at `https://anonymous.4open.science/r/PRISM-DS-KB-4C4D`. We will release the codebase and archival KB publicly upon acceptance.

Limitations. Our current KB is optimized for Python-centered data-science workflows; extending the same taxonomy and skill format to adjacent ecosystems such as R, Julia, and SQL-heavy analytics is a natural direction for future work.

References

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. 2021. Program synthesis with large language models. *Preprint*, arXiv:2108.07732.

Jun Shern Chan, Neil Chowdhury, Oliver Jaffe, James Aung, Dane Sherburn, Evan Mays, Giulio Starace, Kevin Liu, Leon Maksin, Tejal Patwardhan, et al. 2024. Mle-bench: Evaluating machine learning agents on machine learning engineering. *arXiv preprint arXiv:2410.07095*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, and 39 others. 2021. Evaluating large language models trained on code. *Preprint*, arXiv:2107.03374.

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. *Preprint*, arXiv:2510.08511.

Alex Egg, Martin Iglesias Goyanes, Friso Kingma, Andreu Mora, Leandro von Werra, and Thomas Wolf. 2025. Dabstep: Data agent benchmark for multi-step reasoning. *Preprint*, arXiv:2506.23719.

Nick Erickson, Jonas Mueller, Alexander Shirkov, Hang Zhang, Pedro Larroy, Mu Li, and Alexander Smola. 2020. Autogluon-tabular: Robust and accurate automl for structured data. *Preprint*, arXiv:2003.06505.

Siyuan Guo, Cheng Deng, Ying Wen, Hechang Chen, Yi Chang, and Jun Wang. 2024. Ds-agent: Automated data science by empowering large language models with case-based reasoning. *arXiv preprint arXiv:2402.17453*.

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. *Preprint*, arXiv:2502.13138.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. Swe-bench: Can language models resolve real-world github issues? *Preprint*, arXiv:2310.06770.

Liqiang Jing, Zhehui Huang, Xiaoyang Wang, Wenlin Yao, Wenhao Yu, Kaixin Ma, Hongming Zhang, Xinya Du, and Dong Yu. 2024. Dsbench: How far are data science agents from becoming data science experts? *arXiv preprint arXiv:2409.07703*.

Ram Mohan Rao Kadiyala, Jebish Purbey, Siddhant Gupta, Giulio Martini, Suman Debnath, and Hamza Farooq. 2025. DSBC : Data science task benchmarking with context engineering. In *Proceedings of the 14th International Joint Conference on Natural Language Processing and the 4th Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics*, pages 3392–3424, Mumbai, India. The Asian Federation of Natural Language Processing and The Association for Computational Linguistics.

Lars Kotthoff, Chris Thornton, Holger H. Hoos, Frank Hutter, and Kevin Leyton-Brown. 2017. Auto-weka 2.0: Automatic model selection and hyperparameter optimization in weka. *Journal of Machine Learning Research*, 18(25):1–5.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Scott Wen tau Yih, Daniel Fried, Sida Wang, and Tao Yu. 2022. Ds-1000: A natural and reliable benchmark for data science code generation. *Preprint*, arXiv:2211.11501.

Sijie Li, Weiwei Sun, Shanda Li, Ameet Talwalkar, and Yiming Yang. 2026. Comind: Towards community-driven agents for machine learning engineering. *Preprint*, arXiv:2506.20640.

Zexi Liu, Yuzhu Cai, Xinyu Zhu, Yujie Zheng, Runkun Chen, Ying Wen, Yanfeng Wang, Siheng Chen, et al. 2025. Ml-master: Towards ai-for-ai via integration of exploration and reasoning. *arXiv preprint arXiv:2506.16499*.

Jaehyun Nam, Jinsung Yoon, Jiefeng Chen, Jinwoo Shin, Sercan Ö Arık, 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. *Preprint*, arXiv:1603.06212.

Qwen Team. 2025. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language models can teach themselves to use tools. *Preprint*, arXiv:2302.04761.

Serper.dev. 2026. Serper: Google search api for developers. Accessed: 2026-04-01.

Fan Shu, Yite Wang, Ruofan Wu, Boyi Liu, Zhewei Yao, Yuxiong He, and Feng Yan. 2026. Dare-bench: Evaluating modeling and instruction fidelity of llms in data science. *Preprint*, arXiv:2602.24288.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. 2023. Voyager: An open-ended embodied agent with large language models. *Preprint*, arXiv:2305.16291.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. React: Synergizing reasoning and acting in language models. *Preprint*, arXiv:2210.03629.

Dan Zhang, Sining Zhoubian, Min Cai, Fengzu Li, Lekang Yang, Wei Wang, Tianjiao Dong, Ziniu Hu, Jie Tang, and Yisong Yue. 2025. Datascibench: An llm agent benchmark for data science. *Preprint*, arXiv:2502.13897.

Xinyu Zhu, Yuzhu Cai, Zexi Liu, Bingyang Zheng, Cheng Wang, Rui Ye, Yuzhi Zhang, Linfeng Zhang, Weinan E, Siheng Chen, and Yanfeng Wang. 2026. Toward ultra-long-horizon agentic science: Cognitive accumulation for machine learning engineering. *Preprint*, arXiv:2601.10402.

A Core Prompt Templates

The section presents the prompts we used in PRISM-DS

Task profiling prompt.

Task Profiling Prompt: System

```
You are an expert data scientist
    specializing in Kaggle competitions.
Your task is to analyze competition
    descriptions and create a structured
    task profile.

You must identify:
1. Data modality (what type of data)
2. Task type (what ML problem to solve)
3. Data challenges (what makes this
    problem difficult)
4. Target metric (how solutions are
    evaluated)
5. Data context (size, features, special
    considerations)

Be specific and actionable. Your analysis
     will guide solution generation.
```

Task Profiling Prompt: User

```
Analyze the following data science task
    and create a detailed task profile.

## Task Description
{task_description}

## Data Information (if available)
{data_info}

## Instructions
Analyze this task and provide a
    structured profile in the following
    JSON format:

```json
{
    "modality": "<Tabular|CV|NLP|
        Time_Series|Audio|Multimodal>",
    "task_type": "<Binary_Classification|
        Multi_Classification|Regression|
        Ranking|Segmentation|Detection|
        Generation>",
    "target_metric": "<metric name, e.g.,
        AUC, RMSE, F1, MAP@K>",
    "metric_direction": "<higher_better|
        lower_better>",

    "challenges": [
        {
            "name": "<challenge name>",
            "severity": "<high|medium|low
                >",
            "evidence": "<why you
                identified this challenge
                >"
        }
    ],

    "data_context": {
        "estimated_samples": <number or
            null>,
        "estimated_features": <number or
            null>,
        "has_text": <true|false>,
        "has_images": <true|false>,
        "has_time_component": <true|false>,

        "has_categorical": <true|false>,
        "has_numerical": <true|false>,
        "special_considerations": ["<any
            special notes>"]
    },

    "recommended_approaches": [
        "<high-level approach
            recommendation>"
    ],

    "key_insights": [
        "<important insight about the task
            >"
    ]
}
```

Additional requirements (CRITICAL for
    time-series tasks):
- Use the provided data_info to infer the
    train/test split semantics, e.g.:
  - forecast_only: test dates strictly
      after train end per entity
  - resplit_or_interpolation: test
      contains dates earlier than or
      overlapping train end per entity
  - mixed: mixture across entities
- If data_info includes a "Time Split
    Diagnostics (auto)" section, you MUST
     reflect it explicitly in:
  - data_context.special_considerations
```

```
  - key_insights
  - recommended_approaches

## Available Challenge Types
Choose from these standard challenge
    types:
- Imbalanced_Data
- High_Cardinality
- Missing_Values
- Noisy_Labels
- Small_Dataset
- High_Dimensionality
- Temporal_Dependency
- Class_Overlap
- Outliers
- Feature_Leakage
- Multi_Label
- Long_Tail

Provide your analysis:
```

Initial solution prompt.

Initial Solution Prompt: System

```
You are an expert Kaggle grandmaster with
     extensive experience in winning
    competitions.
Your task is to generate a complete, well-
    structured solution for a data
    science competition.

Key principles:
1. Use YOUR OWN expert judgment as the
    primary guide for the solution
2. The Knowledge Base provides OPTIONAL
    reference techniques -- only adopt
    one if it clearly fits this specific
    task
3. It is perfectly fine to IGNORE all KB
    suggestions and use your own approach
4. Optimize for the specified metric
5. Write clean, readable, well-commented
    code -- prefer simplicity over
    complexity

IMPORTANT FILE PATHS:
- Input data files are located in the './
    input/' directory
- Save output/submission files to './
    submission/' directory
- Use './working/' for any intermediate
    files

CRITICAL CODE FORMAT REQUIREMENTS:
- Your code MUST have proper line breaks
    and indentation
- Each statement should be on its own
    line
- Use proper Python formatting with
    newlines between functions and
    classes
- DO NOT output code in a single line

AVAILABLE PACKAGES (ONLY USE THESE):
- scikit-learn, xgboost, lightgbm,
    catboost
- torch (CPU only), tensorflow, keras
- statsmodels, pmdarima, prophet, darts,
    statsforecast
- numpy, pandas, scipy
- tqdm, joblib, pickle
- pillow (PIL), matplotlib, seaborn

NOT AVAILABLE:
- transformers, sentencepiece, tiktoken,
    tokenizers
- GPU-heavy models
- opencv-python (cv2) for most tasks

For NLP/text tasks: use traditional ML
    rather than transformer models.
For tabular tasks: prefer gradient
    boosting.

OUTPUT FORMAT (CRITICAL):
- Return STRICT JSON only
- The JSON must contain:
  - "code": a single Python script as a
      string
  - "summary": one short sentence
      describing the approach
```

Initial Solution Prompt: User

```
Generate a complete solution for this
    data science task.

## Task Profile
{task_profile}

## Optional Reference: Strategy Cards
    from Knowledge Base
{kb_knowledge}

## Additional Information from Online
    Search
{online_knowledge}

## Data Files
{data_files}

## Instructions
Generate a complete Python solution that:
1. Loads all necessary data files from
    './input/'.
2. Cleans the data.
3. Creates features addressing the
    identified challenges.
4. Preprocesses the data for modeling.
5. Defines appropriate model(s).
6. Trains on './input/train.csv' (NO
    multi-fold CV).
7. Generates predictions on test data.
8. Creates './submission/submission.csv'.
9. Produces required validation artifacts.


Important:
{time_split_diagnostics_guidance}
- The code must be self-contained and
    runnable
```

```
- Include all necessary imports at the
    top
- Add comments explaining key decisions
- Handle edge cases gracefully
- Target metric: {target_metric} ({
    metric_direction})
- Do NOT use multi-fold cross-validation
- If non-CSV artifacts exist in ./input/,
     attempt to incorporate them into
    features
- If images are present, choose between
    lightweight feature extraction and a
    CPU-friendly image model depending on
     whether the tabular features are
    already rich
- Preserve row-order alignment in the
    final submission
- Use vectorized operations where
    possible for efficiency

## Output
Return STRICT JSON only with keys:
- "code": string (complete Python script)
- "summary": string (one sentence)
```

Block improvement prompt.

Block Improvement Prompt: System

```
You are an expert ML engineer
    specializing in optimizing ML
    solutions.

Your task is to improve an ML solution,
    focusing on a specific IMPROVEMENT
    DIRECTION.

Key principles:
1. Use your expertise
2. Focus on the actual bottleneck
3. Quality over quantity
4. KB as reference
5. Preserve working parts

IMPORTANT:
- Maximize the target metric
- Think critically about the bottleneck,
    the simplest useful change, obvious
    bugs, and whether a different model
    would be more suitable

CRITICAL CODE FORMAT REQUIREMENTS:
- Proper line breaks and indentation
- Each statement on its own line
- No single-line code dumps

AVAILABLE PACKAGES:
- scikit-learn, xgboost, lightgbm,
    catboost
- torch (CPU only), tensorflow, keras
- statsmodels, pmdarima, prophet, darts,
    statsforecast
- numpy, pandas, scipy

NOT AVAILABLE:
- transformers and related tokenization
    libraries
- GPU-heavy models

For NLP/text tasks: use TF-IDF + LightGBM
    /XGBoost, not transformer models.
```

Block Improvement Prompt: User

```
Improve the following ML solution,
    focusing on the specified direction.

## IMPROVEMENT DIRECTION
Focus Area: {code_stage}

## CURRENT FULL SOLUTION
{full_code}

## CURRENT PERFORMANCE
- Score: {current_score:.6f}
- Target Metric: {target_metric} ({
    metric_direction})

## TASK CONTEXT
{task_context}

## DATA OVERVIEW
{data_overview}

## OPTIONAL REFERENCE: Strategy Cards
    from Knowledge Base
{kb_techniques}

## PREVIOUS IMPROVEMENT ATTEMPTS
{previous_attempts}

## YOUR TASK
Analyze the code and improve it, focusing
     on {code_stage}.

Critical guidance:
1. Identify the actual bottleneck
2. Prefer the simplest change that can
    improve performance
3. KB cards are optional references
4. Avoid unnecessary complexity

Guidelines:
- Return the COMPLETE modified solution
- Prefer targeted changes over large
    rewrites
- Fix obvious bugs first
- Consider whether the model and
    validation strategy are appropriate
- Do NOT use multi-fold cross-validation
- If images are present, avoid metadata-
    only usage when a lightweight image
    component is feasible
- Preserve submission row-order alignment
- Use vectorized operations where
    possible
- Produce required validation artifacts

## OUTPUT FORMAT
Return STRICT JSON only with:
- analysis
- strategy
```

```
- technique_used
- expected_impact
- code
```

Improvement prediction prompt.

Improvement Prediction Prompt: System

```
You are a senior ML engineer evaluating
    proposed code changes.

Your task is to predict whether a code
    change will improve the model's score
    .

Be critical and conservative:
- Code bloat often hurts performance
- Adding features without removing
    ineffective ones rarely helps
- Fundamental algorithmic changes are
    more likely to help than cosmetic
    changes
- More code does not imply better
    performance

Output STRICT JSON only with:
{"improve": true|false, "reason": "one
    short sentence"}
```

Improvement Prediction Prompt: User

```
Predict whether this code change will
    improve the score.

## CHANGE SUMMARY
{change_summary}

## CODE STATISTICS
- Original: {old_lines} lines, {old_chars}
     chars
- Modified: {new_lines} lines, {new_chars}
     chars
- Change: {line_diff:+d} lines ({
    change_pct:+.1f}%)

## CURRENT PERFORMANCE
- Score: {current_score:.6f}
- Metric: {target_metric} ({
    metric_direction})

## KEY DIFFERENCES
{diff_preview}

## HISTORICAL CONTEXT
{previous_attempts}

Based on your ML expertise, will this
    change improve the score?
Be skeptical of large code increases,
    unnecessary feature additions, and
    overly complex changes.

Return STRICT JSON only with:
{"improve": true, "reason": "one short
    sentence"}
```

Debug prompt.

Debug Prompt: System

```
You are an expert Python debugger and
    data scientist.
Your task is to analyze the execution
    error and fix the code.

Key principles:
1. Carefully read the error message and
    traceback
2. Identify the root cause of the error
3. Fix ONLY the problematic code
4. Ensure the fix does not introduce new
    bugs
5. Maintain compatibility with the
    existing code structure

IMPORTANT FILE PATHS:
- Input data files are located in the './
    input/' directory
- Save output/submission files to './
    submission/' directory
- Use './working/' for any intermediate
    files
- Also save validation predictions to './
    validation/validation.csv' (REQUIRED)

CRITICAL CODE FORMAT REQUIREMENTS:
- Proper line breaks and indentation
- Each statement should be on its own
    line
- Do NOT output code in a single line

AVAILABLE PACKAGES:
- scikit-learn, xgboost, lightgbm,
    catboost
- torch (CPU only), tensorflow, keras
- statsmodels, pmdarima, prophet, darts,
    statsforecast
- numpy, pandas, scipy

NOT AVAILABLE:
- transformers, huggingface models,
    sentencepiece, tiktoken, tokenizers
- opencv-python (cv2) for most tasks

If the error is about an unavailable
    package, replace it with an available
     alternative.
```

Debug Prompt: User

```
The following code failed to execute.
    Please fix it.

## Original Code
{code}
```

```
## Execution Error
{error}

## Stdout (if any)
{stdout}

## Debug Attempt
This is debug attempt {attempt} of {
    max_attempts}.

## Instructions
1. Analyze the error carefully
2. Identify the exact cause of the
    failure
3. Fix the code to resolve the error
4. Return the COMPLETE fixed code

Common issues to check:
- Import errors
- Path issues
- API compatibility
- Type errors
- Index errors
- Submission bugs
- Validation artifact requirements

## Output Format
Return STRICT JSON only with:
{"code": "<complete fixed python script>",
     "summary": "<one short sentence>"}
```

Evaluation decision prompt.

Evaluation Decision Prompt: System

```
You are a rigorous ML evaluation judge.
You must decide whether to accept a new
    iteration based on validation scores
    and the described change.

Rules:
- Be conservative: reject changes that
    look like noise, leakage, or metric
    mismatch.
- Consider the metric direction (
    higher_better vs lower_better).
- Output STRICT JSON only.
```

Evaluation Decision Prompt: User

```
Decide whether the new iteration should
    be accepted.

## Context
- Task: {task_name}
- Metric: {metric_name} ({
    metric_direction})

## Scores
- Best validation score so far: {
    best_score}
- Current validation score: {
    current_score}

## Change summary (natural language)
{change_summary}

## Instructions
Return STRICT JSON with:
{
  "improve": true,
  "reason": "one short sentence"
}

Guidance:
- If metric_direction is "higher_better",
     accept when current_score is
    meaningfully higher than best_score.
- If metric_direction is "lower_better",
    accept when current_score is
    meaningfully lower than best_score.
- If best_score is not finite/unknown,
    accept if current_score is finite and
     the change seems reasonable.
```

B Skill KB Distribution

This section provides additional details about the curated skill KB used by PRISM-DS, including its overall scale, taxonomy structure, and the metadata stored for each skill unit.

Table 8: Summary statistics of the curated skill KB.

Statistic	Count
Medal-winning notebooks	1,504
Curated skills	4,421
Modalities	5
Task types	13
Challenge categories	21
Solution families	30
Code stages	13
Full-pipeline recipes	27

Table 9: Taxonomy structure of the curated KB.

Level	Role	Count
Modality	Top-level data domain (e.g., tabular, time series)	5
Task type	Problem formulation within a modality	13
Challenge category	Diagnosed bottleneck or constraint	21
Solution family	Method family addressing the challenge	30
Code stage tag	Pipeline stage used for refinement retrieval	13

Table 10: Pipeline stage inventory used for stage-aware retrieval.

Stage	Purpose
`data_loading`	Read and organize raw inputs
`data_cleaning`	Handle missing values, duplicates, and invalid rows
`eda`	Perform exploratory inspection and diagnostics
`feature_engineering`	Construct new task-relevant features
`feature_selection`	Remove noisy or redundant features
`data_preprocessing`	Apply transforms and encoding before modeling
`model_definition`	Specify model class and architecture
`training_loop`	Fit model parameters
`hyperparameter_tuning`	Adjust model hyperparameters
`validation`	Evaluate on held-out data
`inference`	Generate test-time predictions
`post_processing`	Calibrate or transform predictions
`ensemble`	Combine multiple models or predictions

Table 11: Information stored for each curated skill unit.

Field group	Content
Taxonomy	Modality, task type, challenge category, solution family
Preview card	Method name, problem type, applicability, conditions, expected effect
Implementation block	Strategy description, when-to-use guidance, pseudocode, key parameters, optional code reference
Stage tags	One or more pipeline stages used for stage-aware retrieval during refinement
Source metadata	Origin notebook / competition information for leakage filtering

Operationally, the KB is indexed along the same axes used by profiling, which enables retrieval by challenge and by pipeline stage rather than by free-text matching alone. At inference time, the retrieval agent first uses the task profile to narrow the search space, then reads preview cards for candidate skills, and finally expands selected skills into fuller implementation guidance when needed.

C UCT Block Selection

For block-level refinement, we score each candidate block with a stage-aware UCT criterion:

$$\mathrm{UCT}_i \;=\; \bar{r}_i \;+\; C\sqrt{\frac{\ln V}{n_i}} \;+\; \lambda_{\sigma_i},$$

where $\bar{r}_i$ is the empirical reward of block i, n_i is its visit count, V is the total number of refinement selections so far, C is a decaying exploration constant, and λ_{σ_i} is a stage prior for the stage label σ_i. The first two terms are the standard exploitation–exploration trade-off, while the stage prior biases search toward pipeline stages that are more likely to yield meaningful gains.

D Checkpoint and solution selection

DS-Agent uses the solution generated at the *current* step as the checkpoint evaluated at that step. **ML-Master**, **MLE-STAR**, and **PRISM-DS** each use, at every step, the solution with the highest validation score seen *so far* as the best checkpoint.

E Skill source ablation protocol

All rows in Table 7 use the same GPT-5 agent backend, 10-step budget, and profiling-based pipeline as the main DSBench experiments; only the knowledge delivered into initial generation and refinement (and its retrieval mechanism) changes.

Curated KB (full system). This is the default: profile-guided agentic retrieval over the curated taxonomy-organised skill KB (subsection 3.3).

DS-Agent skills + embedding retrieval. We disable agentic KB retrieval and instead feed the initial generator with text retrieved in one shot from the DS-Agent case corpus (the same `nlp_cases`, `tsa_cases`, and `tabular_cases` text files bundled with DS-Agent's development data). We embed the task description with the `BAAI/llm-embedder` model using the query prefix `Represent this query for retrieving relevant documents:`, L2-normalise embeddings, and retrieve the top-5 cases by cosine similarity in the embedding space, matching DS-Agent's dense retrieval style. Case files whose stem matches the current task slug (same competition) are excluded to avoid trivially retrieving the target task's own write-up.

Online search only. We disable the curated KB and rely on the same online search tool used elsewhere in PRISM-DS (web search with the configured provider and top-k results passed into the prompts), without profile-conditioned agentic browsing of the structured KB.

F Main results metrics on DSBench

Table 12: Step-wise Relative Performance Gap (RPG, %) on DSBench. Each agent is allowed up to 10 refinement steps. Each entry reports the mean and standard deviation achieved up to that step; bold marks the best result for each model.

(a) Steps S0–S4

Framework	Model	S0	S1	S2	S3	S4
DS-Agent	GPT-5	48.57$_{\pm 2.21}$	51.16$_{\pm 0.63}$	52.41$_{\pm 0.42}$	52.75$_{\pm 0.81}$	53.28$_{\pm 0.90}$
	Sonnet-4.5	52.67$_{\pm 1.01}$	54.97$_{\pm 0.55}$	55.16$_{\pm 0.23}$	55.13$_{\pm 1.24}$	55.25$_{\pm 0.63}$
ML-Master	GPT-5	39.30$_{\pm 4.46}$	53.82$_{\pm 2.37}$	55.10$_{\pm 0.84}$	55.00$_{\pm 0.47}$	54.10$_{\pm 0.77}$
	Sonnet-4.5	54.20$_{\pm 1.09}$	55.40$_{\pm 0.22}$	55.11$_{\pm 0.49}$	55.99$_{\pm 0.58}$	55.87$_{\pm 0.68}$
MLE-STAR	GPT-5	47.35$_{\pm 2.94}$	53.55$_{\pm 1.76}$	55.33$_{\pm 0.49}$	53.33$_{\pm 0.57}$	54.19$_{\pm 0.86}$
	Sonnet-4.5	42.48$_{\pm 2.24}$	55.24$_{\pm 1.52}$	56.03$_{\pm 1.01}$	56.71$_{\pm 1.20}$	55.03$_{\pm 0.44}$
PRISM-DS (Ours)	GPT-5	**58.06**$_{\pm 2.01}$	**62.07**$_{\pm 0.52}$	**62.93**$_{\pm 0.09}$	**63.23**$_{\pm 0.28}$	**63.38**$_{\pm 0.30}$
	Sonnet-4.5	**57.82**$_{\pm 0.11}$	**59.29**$_{\pm 0.85}$	**59.08**$_{\pm 0.81}$	**59.09**$_{\pm 0.90}$	**59.10**$_{\pm 0.75}$

(b) Steps S5–S9

Framework	Model	S5	S6	S7	S8	S9
DS-Agent	GPT-5	53.71$_{\pm 0.80}$	54.29$_{\pm 1.01}$	54.85$_{\pm 0.97}$	54.88$_{\pm 0.86}$	54.99$_{\pm 0.66}$
	Sonnet-4.5	55.31$_{\pm 0.92}$	55.64$_{\pm 0.75}$	55.51$_{\pm 0.65}$	55.44$_{\pm 0.60}$	55.87$_{\pm 0.80}$
ML-Master	GPT-5	53.48$_{\pm 0.60}$	54.62$_{\pm 0.59}$	54.55$_{\pm 0.74}$	53.84$_{\pm 1.09}$	53.75$_{\pm 0.92}$
	Sonnet-4.5	55.77$_{\pm 0.58}$	55.96$_{\pm 0.56}$	56.11$_{\pm 0.59}$	55.84$_{\pm 1.03}$	56.14$_{\pm 0.71}$
MLE-STAR	GPT-5	54.49$_{\pm 1.16}$	54.56$_{\pm 1.39}$	54.32$_{\pm 1.29}$	53.89$_{\pm 1.57}$	54.30$_{\pm 1.07}$
	Sonnet-4.5	54.61$_{\pm 0.67}$	54.58$_{\pm 0.53}$	54.62$_{\pm 1.01}$	54.35$_{\pm 0.87}$	54.34$_{\pm 1.23}$
PRISM-DS (Ours)	GPT-5	**63.44**$_{\pm 0.28}$	**63.49**$_{\pm 0.32}$	**63.55**$_{\pm 0.34}$	**63.83**$_{\pm 0.43}$	**64.01**$_{\pm 0.46}$
	Sonnet-4.5	**59.15**$_{\pm 0.82}$	**59.54**$_{\pm 0.71}$	**59.66**$_{\pm 0.73}$	**59.60**$_{\pm 0.72}$	**59.58**$_{\pm 0.69}$

G Open-source-backbone and fixed-time details

Table 13 reports the complete Qwen3-32B trajectory used in the main text.

Table 13: Step-wise RPG (%) on DSBench with Qwen3-32B.

(a) Steps S0–S4

Framework	S0	S1	S2	S3	S4
DS-Agent	0.00	45.73	47.09	49.72	46.74
ML-Master	40.92	47.84	51.84	52.60	53.19
MLE-STAR	0.00	44.35	51.25	52.38	50.81
PRISM-DS (Ours)	**48.47**	**50.75**	**53.05**	**54.03**	**54.73**

(b) Steps S5–S9

Framework	S5	S6	S7	S8	S9
DS-Agent	48.40	47.22	48.54	49.77	48.90
ML-Master	52.17	52.28	51.92	53.05	52.29
MLE-STAR	46.83	47.94	46.43	45.35	45.57
PRISM-DS (Ours)	**54.71**	**55.02**	**55.79**	**55.82**	**55.71**

H Fixed-time evaluation protocol

For the fixed-time results in Table 14, each method starts from the same task inputs and compute environment. At each wall-clock cutoff, we score the best valid checkpoint produced by that time; later checkpoints are unavailable to earlier cutoffs. Across the evaluated tasks, every profile mapped to at least one represented taxonomy branch, so we did not observe a fully out-of-taxonomy case. For such a case, the implemented fallback is open-web search followed by the same relevance filter used for KB candidates; evaluating this fallback on a purpose-built out-of-taxonomy benchmark remains future work.

Table 14: DSBench performance under fixed wall-clock budgets. At each cutoff, we evaluate the best checkpoint produced by that time.

Framework	1h	2h	4h
DS-Agent	0.361	0.477	0.516
ML-Master	0.452	0.528	0.515
MLE-STAR	0.477	0.516	0.538
PRISM-DS	**0.545**	**0.611**	**0.637**

I Case study: final solution code for `liverpool-ion-switching`

This section lists the final Python solutions for the DSBench (Jing et al., 2024) task `liverpool-ion-switching` discussed in subsection 4.4.

PRISM-DS

```
import os
import sys
import gc
import zipfile
import numpy as np
import pandas as pd
from typing import List, Tuple, Dict
from sklearn.preprocessing import LabelEncoder
from sklearn.utils.class_weight import
      compute_class_weight
import lightgbm as lgb
from scipy.signal import savgol_filter

# ============================================
# Utility: Ensure directories
# ============================================

def ensure_dir(path: str) -> None:
    if not os.path.exists(path):
        os.makedirs(path, exist_ok=True)


# ============================================
# Utility: Unzip any archives in ./input/
# ============================================

def unzip_input_archives(input_dir: str = "./input/")
       -> None:
    try:
        for fname in os.listdir(input_dir):
            if fname.lower().endswith('.zip'):
                zip_path = os.path.join(input_dir,
                      fname)
                try:
                    with zipfile.ZipFile(zip_path, 'r')
                           as zf:
                        zf.extractall(input_dir)
                        print(f"Extracted: {zip_path}")
                except Exception as e:
                    print(f"Warning: Failed to extract
                          {zip_path} due to {e}")
    except Exception as e:
        print(f"Warning: unzip_input_archives
              encountered an error: {e}")


# ============================================
# Feature Engineering Helpers
# ============================================
```

```
def safe_mad(x: np.ndarray, c: float = 1.4826, eps:
      float = 1e-6) -> float:
    med = np.median(x)
    mad = np.median(np.abs(x - med))
    mad_scaled = c * mad
    if mad_scaled < eps:
        mad_scaled = eps
    return mad_scaled


def moving_average(arr: np.ndarray, window: int) ->
      np.ndarray:
    n = len(arr)
    if window <= 1 or n == 0:
        return arr.copy()
    w = min(window, n)
    kernel = np.ones(w, dtype=np.float32) / float(w)
    return np.convolve(arr, kernel, mode='same')


def moving_std(arr: np.ndarray, window: int) -> np.
      ndarray:
    n = len(arr)
    if window <= 1 or n == 0:
        return np.zeros_like(arr)
    w = min(window, n)
    kernel = np.ones(w, dtype=np.float32) / float(w)
    mean = np.convolve(arr, kernel, mode='same')
    mean_sq = np.convolve(arr * arr, kernel, mode='
          same')
    var = mean_sq - mean * mean
    var[var < 0] = 0.0
    return np.sqrt(var)


def _effective_sg_params(n: int, desired_window: int,
       max_poly: int = 3) -> Tuple[int, int]:
    # Ensure an odd window length <= n and polyorder <
          window_length
    if n <= 1:
        return 1, 0
    w = min(desired_window, n)
    if w % 2 == 0:
        w = w - 1 if w > 1 else 1
    if w < 3:
        # minimal smoothing possible
        poly = 1 if w > 1 else 0
        return w, poly
    poly = min(max_poly, w - 1)
    if poly < 1:
        poly = 1
    return w, poly


def compute_batch_features(time_vals: np.ndarray,
                           signal_vals: np.ndarray,
                           windows: List[int],
                           sg_windows: List[int] = [11,
                                 51]) -> Dict[str, np.
                                 ndarray]:
    n = len(signal_vals)
    out = {}

    # Robust scaling within the batch
    med = np.median(signal_vals)
    mad = safe_mad(signal_vals)
    s_robust = (signal_vals - med) / mad

    # Standard z-score within batch
    mean = signal_vals.mean()
    std = signal_vals.std()
    if std < 1e-6:
        std = 1e-6
    s_z = (signal_vals - mean) / std

    # Quantile-based scaling within batch (robust to
          outliers)
    try:
        q10, q50, q90 = np.quantile(signal_vals, [0.1,
              0.5, 0.9])
    except Exception:
        q10 = np.percentile(signal_vals, 10)
        q50 = np.percentile(signal_vals, 50)
        q90 = np.percentile(signal_vals, 90)
    iqr = max(q90 - q10, 1e-6)
    s_qnorm = (signal_vals - q50) / iqr
    s_qminmax = (signal_vals - q10) / iqr
    s_qminmax = np.clip(s_qminmax, 0.0, 1.0)

    # Differences using index-domain and fix alignment
    diff1 = np.zeros(n, dtype=np.float32)
    if n > 1:
        diff1[1:] = np.diff(signal_vals)
    diff2 = np.zeros(n, dtype=np.float32)
    if n > 2:
        # second difference aligned starting at
              position 2
        diff2[2:] = np.diff(signal_vals, n=2)

    # Time-aware derivatives via Savitzky-Golay
          smoothing
    if n > 1:
        dt = np.median(np.diff(time_vals))
        if not np.isfinite(dt) or dt <= 0:
            dt = 1.0
    else:
        dt = 1.0

    for sw in sg_windows:
        eff_w, poly = _effective_sg_params(n, sw,
              max_poly=3)
        if eff_w >= 3 and poly >= 1 and eff_w <= n:
            try:
                sg0 = savgol_filter(signal_vals,
                      window_length=eff_w, polyorder=
                      poly, deriv=0, delta=dt, mode='
                      interp')
                sg1 = savgol_filter(signal_vals,
                      window_length=eff_w, polyorder=
                      poly, deriv=1, delta=dt, mode='
                      interp')
                sg2 = savgol_filter(signal_vals,
                      window_length=eff_w, polyorder=
                      poly, deriv=2, delta=dt, mode='
                      interp')
            except Exception:
                # Fallback to zeros if scipy fails
                sg0 = np.zeros(n, dtype=np.float32)
                sg1 = np.zeros(n, dtype=np.float32)
                sg2 = np.zeros(n, dtype=np.float32)
        else:
            sg0 = np.zeros(n, dtype=np.float32)
            sg1 = np.zeros(n, dtype=np.float32)
            sg2 = np.zeros(n, dtype=np.float32)
        out[f'sg_smooth_{sw}'] = sg0.astype(np.float32
              )
        out[f'sg_d1_{sw}'] = sg1.astype(np.float32)
        out[f'sg_d2_{sw}'] = sg2.astype(np.float32)

    # Lags and leads on raw signal
    def lag(arr: np.ndarray, k: int) -> np.ndarray:
        out_l = np.roll(arr, k)
        if k > 0:
            out_l[:k] = 0.0
        return out_l

    def lead(arr: np.ndarray, k: int) -> np.ndarray:
        out_l = np.roll(arr, -k)
        if k > 0:
            out_l[-k:] = 0.0
        return out_l

    out['s_raw'] = signal_vals.astype(np.float32)
    out['s_robust'] = s_robust.astype(np.float32)
    out['s_z'] = s_z.astype(np.float32)
    out['s_qnorm'] = np.clip(s_qnorm, -8.0, 8.0).
          astype(np.float32)
    out['s_qminmax'] = s_qminmax.astype(np.float32)
    out['diff1'] = diff1.astype(np.float32)
    out['diff2'] = diff2.astype(np.float32)

    # Rolling features (centered) on robust signal and
          raw signal
    for w in windows:
        ma_r = moving_average(s_robust, w).astype(np.
              float32)
        ms_r = moving_std(s_robust, w).astype(np.
              float32)
        out[f'rob_ma_{w}'] = ma_r
```

```
        out[f'rob_ms_{w}'] = ms_r
        ma_raw = moving_average(signal_vals, w).astype
             (np.float32)
        ms_raw = moving_std(signal_vals, w).astype(np.
             float32)
        out[f'raw_ma_{w}'] = ma_raw
        out[f'raw_ms_{w}'] = ms_raw

    # Lags/leads for local context (uses only signal,
         no labels)
    for k in [1, 2, 3]:
        out[f'lag_{k}'] = lag(signal_vals, k).astype(
             np.float32)
        out[f'lead_{k}'] = lead(signal_vals, k).astype
             (np.float32)

    # Clip some heavy-tailed features for robustness
    out['s_robust'] = np.clip(out['s_robust'], -8.0,
         8.0)
    out['s_z'] = np.clip(out['s_z'], -8.0, 8.0)

    return out


def build_features(df: pd.DataFrame,
                   windows: List[int] = [5, 25, 100,
                        500],
                   sg_windows: List[int] = [11, 51]) ->
                         pd.DataFrame:
    # Preserve original order and ensure a clean
         positional index
    df = df.copy()
    df.reset_index(drop=True, inplace=True)
    df['__row_id__'] = np.arange(len(df))

    # Define batch based on known 50-second segments;
         do not cross batches
    df['batch'] = np.floor(df['time'] / 50.0).astype(
         int)
    # Position within batch
    df['batch_pos'] = df['time'] - df['batch'] * 50.0
    df['batch_pos_rel'] = (df['batch_pos'] / 50.0).
         astype(np.float32)

    # Prepare list of feature columns
    feature_cols = ['s_raw', 's_robust', 's_z', '
         s_qnorm', 's_qminmax', 'diff1', 'diff2', '
         batch_pos', 'batch_pos_rel']
    for w in windows:
        feature_cols += [f'rob_ma_{w}', f'rob_ms_{w}',
             f'raw_ma_{w}', f'raw_ms_{w}']
    for k in [1, 2, 3]:
        feature_cols += [f'lag_{k}', f'lead_{k}']
    for sw in sg_windows:
        feature_cols += [f'sg_smooth_{sw}', f'sg_d1_{
             sw}', f'sg_d2_{sw}']

    # Create feature DataFrame aligned to df index
    feat_df = pd.DataFrame(index=df.index)

    # Pre-create columns to simplify assignment
    for col in feature_cols:
        feat_df[col] = np.zeros(len(df), dtype=np.
             float32)

    # Process per batch to avoid leakage
    batches = df['batch'].unique()
    for b in batches:
        mask = df['batch'] == b
        idx_sorted_labels = df.loc[mask].sort_values('
             time').index.values.astype(int)
        time_vals = df.loc[idx_sorted_labels, 'time'].
             to_numpy(dtype=np.float32)
        signal_vals = df.loc[idx_sorted_labels, '
             signal'].to_numpy(dtype=np.float32)

        # Compute batch features
        batch_feats = compute_batch_features(time_vals
             , signal_vals, windows, sg_windows=
             sg_windows)

        # Assign computed features by exact indices to
              ensure shape alignment
        for key, arr in batch_feats.items():
            arr = np.asarray(arr, dtype=np.float32)
            if arr.ndim == 1:
                arr1d = arr.reshape(-1, 1)
            else:
                arr1d = arr
            if arr1d.shape[0] != len(idx_sorted_labels)
                 :
                if arr1d.shape[0] > len(
                     idx_sorted_labels):
                    arr1d = arr1d[:len(
                         idx_sorted_labels), :]
                else:
                    pad_len = len(idx_sorted_labels) -
                         arr1d.shape[0]
                    pad_block = np.zeros((pad_len,
                         arr1d.shape[1]), dtype=np.
                         float32)
                    arr1d = np.vstack([arr1d, pad_block
                         ])
            if key not in feat_df.columns:
                feat_df[key] = np.zeros(len(df), dtype=
                     np.float32)
            feat_df.loc[idx_sorted_labels, [key]] =
                 arr1d

        # Also assign batch position raw features
        bp = df.loc[idx_sorted_labels, 'batch_pos'].
             to_numpy(dtype=np.float32).reshape(-1,
             1)
        bpr = df.loc[idx_sorted_labels, 'batch_pos_rel
             '].to_numpy(dtype=np.float32).reshape(-1,
              1)
        feat_df.loc[idx_sorted_labels, ['batch_pos']]
             = bp
        feat_df.loc[idx_sorted_labels, ['batch_pos_rel
             ']] = bpr

    # Ensure all feature columns exist and proper
         dtype
    for col in feature_cols:
        if col not in feat_df.columns:
            feat_df[col] = np.zeros(len(df), dtype=np.
                 float32)
        feat_df[col] = feat_df[col].astype(np.float32).
             fillna(0.0)

    feat_df['__row_id__'] = df['__row_id__'].values
    feat_df['batch'] = df['batch'].astype(np.int16).
         values

    # Restore original order
    feat_df = feat_df.sort_values('__row_id__').
         reset_index(drop=True)

    # Cleanup
    df.drop(columns=['__row_id__'], inplace=True)

    return feat_df


# ==========================================
# Probability smoothing over time per batch
# ==========================================

def smooth_probabilities_by_batch(df_ref: pd.
     DataFrame,
                                  probs: np.ndarray,
                                  batch_col: str = '
                                       batch',
                                  time_col: str = 'time
                                       ',
                                  window: int = 9) -> np
                                       .ndarray:
    if window < 1:
        return probs

    smoothed = np.zeros_like(probs)
    classes = probs.shape[1]

    df_tmp = pd.DataFrame({
        '__row_id__': np.arange(len(df_ref)),
        batch_col: df_ref[batch_col].values,
        time_col: df_ref[time_col].values
    })

    for b in df_tmp[batch_col].unique():
```

```
        mask = (df_tmp[batch_col] == b)
        idx = np.where(mask)[0]
        if len(idx) == 0:
            continue
        order = np.argsort(df_tmp.loc[mask, time_col].
              values)
        idx_sorted = idx[order]

        p_batch = probs[idx_sorted, :]
        eff_window = min(window, len(idx_sorted))
        if eff_window % 2 == 0:
            eff_window = eff_window + 1 if eff_window
                  < len(idx_sorted) else eff_window - 1
                   if eff_window > 1 else 1
        eff_window = max(1, eff_window)
        kernel = np.ones(eff_window, dtype=np.float32)
               / float(eff_window)
        p_smooth = np.zeros_like(p_batch)
        for c in range(classes):
            conv = np.convolve(p_batch[:, c], kernel,
                  mode='same')
            p_smooth[:, c] = conv
        smoothed[idx_sorted, :] = p_smooth

    return smoothed


# ================================================
# Main Pipeline
# ================================================

def main():
    ensure_dir('./submission')
    ensure_dir('./validation')
    ensure_dir('./working')

    unzip_input_archives('./input')

    # Load data
    train_path_full = './input/train_full.csv'
    train_path = './input/train.csv'
    if os.path.exists(train_path_full):
        train_df = pd.read_csv(train_path_full)
        print('Loaded train_full.csv')
    else:
        train_df = pd.read_csv(train_path)
        print('Loaded train.csv')

    test_df = pd.read_csv('./input/test.csv')
    print('Loaded test.csv')

    valid_df = None
    valid_path = './input/validation.csv'
    if os.path.exists(valid_path):
        valid_df = pd.read_csv(valid_path)
        print('Loaded validation.csv')
    else:
        print('No validation.csv provided in input;
              will skip validation predictions.')

    # Basic cleaning: drop exact duplicate rows if any
           (rare)
    before = len(train_df)
    train_df = train_df.drop_duplicates()
    after = len(train_df)
    if after != before:
        print(f"Dropped {before - after} duplicate
              rows from train")

    # Ensure dtypes
    for col in ['time', 'signal']:
        if col in train_df.columns:
            train_df[col] = train_df[col].astype(np.
                  float32)
        if col in test_df.columns:
            test_df[col] = test_df[col].astype(np.
                  float32)
        if valid_df is not None and col in valid_df.
              columns:
            valid_df[col] = valid_df[col].astype(np.
                  float32)

    # Feature engineering
    print('Building features for train...')
```

```
    train_feat = build_features(train_df[['time', '
          signal']])

    print('Building features for test...')
    test_feat = build_features(test_df[['time', '
          signal']])

    if valid_df is not None:
        print('Building features for validation...')
        valid_feat = build_features(valid_df[['time',
              'signal']])
    else:
        valid_feat = None

    # Prepare labels
    y = train_df['open_channels'].values
    le = LabelEncoder()
    y_enc = le.fit_transform(y)
    num_classes = len(le.classes_)
    print(f"Detected {num_classes} classes: {list(le.
          classes_)}")

    # Assemble training matrix
    feature_cols = [c for c in train_feat.columns if c
           not in ['__row_id__', 'batch']]
    X_train = train_feat[feature_cols].values.astype(
          np.float32)

    # Class weights to counter imbalance (macro F1
          friendly)
    try:
        cls_weights = compute_class_weight(
              class_weight='balanced', classes=np.
              arange(num_classes), y=y_enc)
        class_weight_dict = {i: float(w) for i, w in
              enumerate(cls_weights)}
    except Exception:
        class_weight_dict = 'balanced'

    # Define model (single model, no CV per
          instructions)
    print('Training LightGBM...')
    model = lgb.LGBMClassifier(
        objective='multiclass',
        num_class=num_classes,
        n_estimators=800,
        learning_rate=0.05,
        num_leaves=128,
        max_depth=-1,
        subsample=0.8,
        colsample_bytree=0.8,
        min_child_samples=20,
        reg_alpha=0.1,
        reg_lambda=0.1,
        random_state=42,
        n_jobs=-1,
        class_weight=class_weight_dict
    )

    model.fit(X_train, y_enc)

    # Free memory
    del X_train
    gc.collect()

    # Inference on test
    print('Predicting on test...')
    X_test = test_feat[feature_cols].values.astype(np.
          float32)
    test_probs = model.predict_proba(X_test)

    # Smooth probabilities per batch
    test_feat_with_meta = test_feat.copy()
    test_feat_with_meta['time'] = test_df['time'].
          astype(np.float32).values
    test_probs_smooth = smooth_probabilities_by_batch(
        df_ref=test_feat_with_meta.assign(batch=
              test_feat['batch']),
        probs=test_probs,
        batch_col='batch',
        time_col='time',
        window=9
    )
```

```
    test_pred_enc = np.argmax(test_probs_smooth, axis
        =1)
    test_pred = le.inverse_transform(test_pred_enc)
    test_pred = test_pred.astype(int)

    # Prepare submission
    sample_sub_path = './input/sample_submission.csv'
    if os.path.exists(sample_sub_path):
        sample_sub = pd.read_csv(sample_sub_path)
        sub_cols = list(sample_sub.columns)
        target_col = 'open_channels' if 'open_channels
            ' in sub_cols else sub_cols[-1]
        id_cols = [c for c in sub_cols if c !=
            target_col]
    else:
        id_cols = ['time']
        target_col = 'open_channels'
        sub_cols = id_cols + [target_col]

    submission_df = pd.DataFrame()
    for c in id_cols:
        if c in test_df.columns:
            submission_df[c] = test_df[c].values
        else:
            if c.lower() == 'id':
                submission_df[c] = np.arange(len(
                    test_df))
    submission_df[target_col] = test_pred
    submission_df = submission_df[sub_cols]

    ensure_dir('./submission')
    submission_path = './submission/submission.csv'
    submission_df.to_csv(submission_path, index=False)
    print(f'Saved submission to {submission_path} with
        shape {submission_df.shape}')

    # Inference on validation (required artifact)
    if valid_df is not None and valid_feat is not None
        :
        print('Predicting on validation...')
        X_valid = valid_feat[feature_cols].values.
            astype(np.float32)
        valid_probs = model.predict_proba(X_valid)

        valid_feat_with_meta = valid_feat.copy()
        valid_feat_with_meta['time'] = valid_df['time
            '].astype(np.float32).values
        valid_probs_smooth =
            smooth_probabilities_by_batch(
            df_ref=valid_feat_with_meta.assign(batch=
                valid_feat['batch']),
            probs=valid_probs,
            batch_col='batch',
            time_col='time',
            window=9
        )
        valid_pred_enc = np.argmax(valid_probs_smooth,
            axis=1)
        valid_pred = le.inverse_transform(
            valid_pred_enc).astype(int)

        validation_out = pd.DataFrame()
        for c in id_cols:
            if c in valid_df.columns:
                validation_out[c] = valid_df[c].values
            else:
                if c.lower() == 'id':
                    validation_out[c] = np.arange(len(
                        valid_df))
        validation_out[target_col] = valid_pred
        validation_out = validation_out[sub_cols]

        ensure_dir('./validation')
        val_out_path = './validation/validation.csv'
        validation_out.to_csv(val_out_path, index=
            False)
        print(f'Saved validation predictions to {
            val_out_path} with shape {validation_out.
            shape}')

    print('Done.')


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

DS-Agent

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import SGDClassifier
from sklearn.pipeline import Pipeline
from sklearn.metrics import f1_score

"""
Training script for liverpool-ion-switching

This is a baseline template. The DS-Agent will modify this script
to implement the solution for the task.
"""

# Load data
train_df = pd.read_csv("train.csv")
test_df = pd.read_csv("test.csv")

print("Train shape:", train_df.shape)
print("Test shape:", test_df.shape)
print("\nTrain columns:", train_df.columns.tolist())
print("\nFirst few rows of train data:")
print(train_df.head())

# Feature engineering: add previous and next signal within continuous segments
def add_neighbor_features(df: pd.DataFrame) -> pd.DataFrame:
    # Determine segment boundaries where time difference exceeds 0.0001
    time_diff = df['time'].diff()
    new_segment = (time_diff > 0.0001) | (time_diff.isna())
    segment_id = new_segment.cumsum().astype(int)

    # Compute previous and next signal within each segment
    df = df.copy()
    df['_segment_id'] = segment_id
    df['signal_prev'] = df.groupby('_segment_id')['signal'].shift(1)
    df['signal_next'] = df.groupby('_segment_id')['signal'].shift(-1)

    # Fill NaNs at segment edges with current signal value
    df['signal_prev'] = df['signal_prev'].fillna(df['signal'])
    df['signal_next'] = df['signal_next'].fillna(df['signal'])

    # Clean up temporary column
    df.drop(columns=['_segment_id'], inplace=True)
    return df

# Apply feature engineering to both train and test dataframes
train_df = add_neighbor_features(train_df)
test_df = add_neighbor_features(test_df)

# Prepare data using the three features
X = train_df[['signal', 'signal_prev', 'signal_next']].values
y = train_df['open_channels'].values

# Fixed validation split (must remain unchanged in future experiments)
X_train, X_val, y_train, y_val = train_test_split(
    X, y, test_size=0.2, shuffle=False, random_state=42
)

# Model: Pipeline with StandardScaler and SGDClassifier
pipeline = Pipeline([
    ('scaler', StandardScaler()),
    ('clf', SGDClassifier(loss='log_loss', class_weight='balanced', average=True, alpha=1e-5, max_iter=1000, random_state=42))
])

# Train and evaluate on validation split
pipeline.fit(X_train, y_train)
y_pred_val = pipeline.predict(X_val)
macro_f1 = f1_score(y_val, y_pred_val, average='macro')
print(f"\nValidation Macro F1 score: {macro_f1:.6f}")

# Final training on full data
pipeline.fit(X, y)

# Predict on test data
X_test = test_df[['signal', 'signal_prev', 'signal_next']].values
test_preds = pipeline.predict(X_test)

# Create submission.csv with columns ['time', 'open_channels']
submission = pd.DataFrame({
    'time': test_df['time'],
    'open_channels': test_preds
})
submission.to_csv("submission.csv", index=False)

print("\nBaseline script completed. Submission saved to submission.csv.")
```

ML-Master

```
import numpy as np
import pandas as pd
import torch
import torch.nn as nn
from torch.utils.data import Dataset, DataLoader
from sklearn.metrics import f1_score
from sklearn.preprocessing import StandardScaler
from sklearn.utils.class_weight import compute_class_weight
import os
from tqdm import tqdm

# Set device to CPU
device = torch.device("cpu")
print(f"Using device: {device}")

# Create submission directory
os.makedirs("./submission", exist_ok=True)
os.makedirs("./working", exist_ok=True)

# Load data
print("Loading data...")
train_df = pd.read_csv("./input/train.csv")
test_df = pd.read_csv("./input/test.csv")

print(f"Train shape: {train_df.shape}")
print(f"Test shape: {test_df.shape}")


# Feature engineering function with windowing for CNN
def create_windowed_features(df, window_size=50, is_train=True):
    signal = df["signal"].values

    # Create basic statistical features
    features = []
    features.append(signal.reshape(-1, 1))

    # Rolling statistics
    for window in [10, 50, 100]:
        rolling_mean = (
            df["signal"].rolling(window=window, center=True).mean().bfill().ffill()
        )
        rolling_std = (
            df["signal"].rolling(window=window, center=True).std().bfill().ffill()
        )
        features.append(rolling_mean.values.reshape(-1, 1))
        features.append(rolling_std.values.reshape(-1, 1))
```

```
    # Signal differences
    diff1 = np.diff(signal, prepend=signal[0])
    features.append(diff1.reshape(-1, 1))

    X_basic = np.hstack(features)

    # Create windowed sequences for CNN
    n_samples = len(signal)
    half_window = window_size // 2

    # Pad signal for windowing
    padded_signal = np.pad(signal, (half_window,
          half_window), mode="edge")

    # Create windows
    X_windowed = np.zeros((n_samples, window_size))
    for i in range(n_samples):
        X_windowed[i] = padded_signal[i : i +
              window_size]

    if is_train:
        y = df["open_channels"].values
        return X_windowed, X_basic, y
    else:
        return X_windowed, X_basic


print("Creating features for training data...")
window_size = 50
X_train_windowed_full, X_train_basic_full,
      y_train_full = create_windowed_features(
    train_df, window_size=window_size, is_train=True
)

print(f"Windowed features shape: {
      X_train_windowed_full.shape}")
print(f"Basic features shape: {X_train_basic_full.
      shape}")

# Split data - use last 10% for validation
train_size = int(0.9 * len(X_train_windowed_full))
X_train_windowed, X_val_windowed = (
    X_train_windowed_full[:train_size],
    X_train_windowed_full[train_size:],
)
X_train_basic, X_val_basic = (
    X_train_basic_full[:train_size],
    X_train_basic_full[train_size:],
)
y_train, y_val = y_train_full[:train_size],
      y_train_full[train_size:]

print(f"Train size: {X_train_windowed.shape}, Val
      size: {X_val_windowed.shape}")

# Normalize features
scaler_basic = StandardScaler()
X_train_basic = scaler_basic.fit_transform(
      X_train_basic)
X_val_basic = scaler_basic.transform(X_val_basic)

scaler_windowed = StandardScaler()
X_train_windowed = scaler_windowed.fit_transform(
      X_train_windowed)
X_val_windowed = scaler_windowed.transform(
      X_val_windowed)


# Dataset class
class SignalDataset(Dataset):
    def __init__(self, X_windowed, X_basic, y=None):
        self.X_windowed = torch.FloatTensor(X_windowed
              ).unsqueeze(
            1
        ) # Add channel dimension
        self.X_basic = torch.FloatTensor(X_basic)
        self.y = torch.LongTensor(y) if y is not None
              else None

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

    def __getitem__(self, idx):
        if self.y is not None:
            return self.X_windowed[idx], self.X_basic[
                idx], self.y[idx]
        return self.X_windowed[idx], self.X_basic[idx]


# CNN-based Model
class CNNSignalClassifier(nn.Module):
    def __init__(self, window_size,
          basic_features_size, num_classes=11):
        super(CNNSignalClassifier, self).__init__()

        # CNN branch for windowed signal
        self.conv1 = nn.Conv1d(1, 32, kernel_size=5,
              padding=2)
        self.bn1 = nn.BatchNorm1d(32)
        self.pool1 = nn.MaxPool1d(2)

        self.conv2 = nn.Conv1d(32, 64, kernel_size=5,
              padding=2)
        self.bn2 = nn.BatchNorm1d(64)
        self.pool2 = nn.MaxPool1d(2)

        self.conv3 = nn.Conv1d(64, 128, kernel_size=3,
              padding=1)
        self.bn3 = nn.BatchNorm1d(128)

        # Calculate flattened size after convolutions
        conv_output_size = (window_size // 4) * 128

        # FC layers for CNN features
        self.fc_cnn = nn.Linear(conv_output_size, 256)
        self.bn_cnn = nn.BatchNorm1d(256)
        self.dropout_cnn = nn.Dropout(0.3)

        # FC layers for basic features
        self.fc_basic1 = nn.Linear(basic_features_size
              , 128)
        self.bn_basic1 = nn.BatchNorm1d(128)
        self.dropout_basic = nn.Dropout(0.3)

        # Combined layers
        self.fc_combined1 = nn.Linear(256 + 128, 256)
        self.bn_combined1 = nn.BatchNorm1d(256)
        self.dropout_combined1 = nn.Dropout(0.3)

        self.fc_combined2 = nn.Linear(256, 128)
        self.bn_combined2 = nn.BatchNorm1d(128)
        self.dropout_combined2 = nn.Dropout(0.2)

        self.fc_out = nn.Linear(128, num_classes)
        self.relu = nn.ReLU()

    def forward(self, x_windowed, x_basic):
        # CNN branch
        x = self.conv1(x_windowed)
        x = self.bn1(x)
        x = self.relu(x)
        x = self.pool1(x)

        x = self.conv2(x)
        x = self.bn2(x)
        x = self.relu(x)
        x = self.pool2(x)

        x = self.conv3(x)
        x = self.bn3(x)
        x = self.relu(x)

        # Flatten
        x = x.view(x.size(0), -1)

        # FC for CNN features
        x = self.fc_cnn(x)
        x = self.bn_cnn(x)
        x = self.relu(x)
        x = self.dropout_cnn(x)

        # Basic features branch
        x_b = self.fc_basic1(x_basic)
        x_b = self.bn_basic1(x_b)
        x_b = self.relu(x_b)
        x_b = self.dropout_basic(x_b)

        # Combine
        x_combined = torch.cat([x, x_b], dim=1)
```

```
        x_combined = self.fc_combined1(x_combined)
        x_combined = self.bn_combined1(x_combined)
        x_combined = self.relu(x_combined)
        x_combined = self.dropout_combined1(x_combined
            )

        x_combined = self.fc_combined2(x_combined)
        x_combined = self.bn_combined2(x_combined)
        x_combined = self.relu(x_combined)
        x_combined = self.dropout_combined2(x_combined
            )

        out = self.fc_out(x_combined)
        return out


# Create datasets and loaders
train_dataset = SignalDataset(X_train_windowed,
    X_train_basic, y_train)
val_dataset = SignalDataset(X_val_windowed,
    X_val_basic, y_val)

batch_size = 2048
train_loader = DataLoader(
    train_dataset, batch_size=batch_size, shuffle=True
        , num_workers=4
)
val_loader = DataLoader(
    val_dataset, batch_size=batch_size, shuffle=False,
        num_workers=4
)

# Calculate class weights
class_weights = compute_class_weight("balanced",
    classes=np.unique(y_train), y=y_train)
class_weights = torch.FloatTensor(class_weights).to(
    device)

# Initialize model
model = CNNSignalClassifier(
    window_size=window_size, basic_features_size=
        X_train_basic.shape[1], num_classes=11
)
model = model.to(device)

criterion = nn.CrossEntropyLoss(weight=class_weights)
optimizer = torch.optim.Adam(model.parameters(), lr
    =0.001)
scheduler = torch.optim.lr_scheduler.
    ReduceLROnPlateau(
    optimizer, mode="max", factor=0.5, patience=2
)


# Training function
def train_epoch(model, loader, criterion, optimizer):
    model.train()
    total_loss = 0

    for X_windowed_batch, X_basic_batch, y_batch in
        loader:
        X_windowed_batch = X_windowed_batch.to(device)
        X_basic_batch = X_basic_batch.to(device)
        y_batch = y_batch.to(device)

        optimizer.zero_grad()
        outputs = model(X_windowed_batch,
            X_basic_batch)
        loss = criterion(outputs, y_batch)
        loss.backward()
        optimizer.step()

        total_loss += loss.item()

    return total_loss / len(loader)


# Validation function
def validate(model, loader):
    model.eval()
    all_preds = []
    all_labels = []

    with torch.no_grad():
        for X_windowed_batch, X_basic_batch, y_batch
            in loader:
            X_windowed_batch = X_windowed_batch.to(
                device)
            X_basic_batch = X_basic_batch.to(device)

            outputs = model(X_windowed_batch,
                X_basic_batch)
            preds = torch.argmax(outputs, dim=1)

            all_preds.extend(preds.cpu().numpy())
            all_labels.extend(y_batch.numpy())

    # Calculate macro F1
    f1_scores = []
    for class_id in range(11):
        y_true_binary = (np.array(all_labels) ==
            class_id).astype(int)
        y_pred_binary = (np.array(all_preds) ==
            class_id).astype(int)

        if y_true_binary.sum() > 0:
            f1 = f1_score(y_true_binary, y_pred_binary,
                zero_division=0)
            f1_scores.append(f1)

    macro_f1 = np.mean(f1_scores)
    return macro_f1, all_preds


# Training loop
print("\nTraining model...")
num_epochs = 12
best_f1 = 0

for epoch in range(num_epochs):
    train_loss = train_epoch(model, train_loader,
        criterion, optimizer)
    val_f1, _ = validate(model, val_loader)

    scheduler.step(val_f1)

    print(
        f"Epoch {epoch+1}/{num_epochs}, Loss: {
            train_loss:.4f}, Val Macro F1: {val_f1
            :.4f}"
    )

    if val_f1 > best_f1:
        best_f1 = val_f1
        torch.save(model.state_dict(), "./working/
            best_model.pth")

print(f"\nBest Validation Macro F1 Score: {best_f1:.4
    f}")

# Load best model
model.load_state_dict(torch.load("./working/
    best_model.pth"))

# Create test features
print("\nCreating test features...")
X_test_windowed, X_test_basic =
    create_windowed_features(
    test_df, window_size=window_size, is_train=False
)
X_test_basic = scaler_basic.transform(X_test_basic)
X_test_windowed = scaler_windowed.transform(
    X_test_windowed)

# Make predictions
print("Making predictions on test set...")
test_dataset = SignalDataset(X_test_windowed,
    X_test_basic)
test_loader = DataLoader(
    test_dataset, batch_size=batch_size, shuffle=False
        , num_workers=4
)

model.eval()
test_preds = []

with torch.no_grad():
    for X_windowed_batch, X_basic_batch in tqdm(
        test_loader, desc="Predicting"):
```

```
        X_windowed_batch = X_windowed_batch.to(device)
        X_basic_batch = X_basic_batch.to(device)
        outputs = model(X_windowed_batch,
              X_basic_batch)
        preds = torch.argmax(outputs, dim=1)
        test_preds.extend(preds.cpu().numpy())

# Create submission
submission = pd.DataFrame({"time": test_df["time"], "
      open_channels": test_preds})

submission.to_csv("./submission/submission.csv",
      index=False)
print("\nSubmission file created successfully!")
print(f"Submission shape: {submission.shape}")
print(f"Unique predictions: {sorted(submission['
      open_channels'].unique())}")
```

MLE-STAR

```
import os
import numpy as np
import pandas as pd
from catboost import CatBoostClassifier, Pool

# Paths
train_path = './input/train.csv'
test_path = './input/test.csv'
final_dir = './final'
os.makedirs(final_dir, exist_ok=True)

# Load data
train = pd.read_csv(train_path)
test = pd.read_csv(test_path)

# Basic preprocessing
train['time'] = train['time'].astype(np.float64)
train['signal'] = train['signal'].astype(np.float32)
train['open_channels'] = train['open_channels'].
      astype(np.int32)
test['time'] = test['time'].astype(np.float64)
test['signal'] = test['signal'].astype(np.float32)

# Compute batch ids (discrete 50-second segments as
      per description)
train['batch'] = (train['time'] // 50).astype(np.
      int32)
test['batch'] = (test['time'] // 50).astype(np.int32)

# Feature engineering function
def make_features(df):
    df = df.copy()
    df['sig'] = df['signal'].astype(np.float32)

    # Lags within batch
    for s in [1, 2, 3]:
        df[f'lag_{s}'] = df.groupby('batch')['sig'].
              shift(s).astype(np.float32)

    # First difference within batch
    df['diff1'] = df.groupby('batch')['sig'].diff().
          fillna(0).astype(np.float32)

    # Rolling features within batch
    for w in [10, 50, 100]:
        r = (
            df.groupby('batch')['sig']
              .rolling(w, min_periods=1)
              .agg(['mean', 'std', 'min', 'max'])
              .reset_index(level=0, drop=True)
        )
        df[f'rm_{w}'] = r['mean'].astype(np.float32)
        df[f'rs_{w}'] = r['std'].fillna(0).astype(np.
              float32)
        df[f'rmin_{w}'] = r['min'].astype(np.float32)
        df[f'rmax_{w}'] = r['max'].astype(np.float32)

    # Fill any remaining NaNs from lags
    for s in [1, 2, 3]:
        df[f'lag_{s}'] = df[f'lag_{s}'].fillna(0).
              astype(np.float32)

    # Positional feature within batch
    df['pos_in_batch'] = df.groupby('batch').cumcount
          ().astype(np.int32)
    df['pos_in_batch_norm'] = (df['pos_in_batch'] /
          500000.0).astype(np.float32)

    return df

# Build features on full training data
train_fe = make_features(train)

# Prepare feature columns
drop_cols = ['time', 'open_channels']
feature_cols = [c for c in train_fe.columns if c not
      in drop_cols]
# Ensure batch is categorical feature
cat_features = [feature_cols.index('batch')] if '
      batch' in feature_cols else []

# Targets
y_train = train_fe['open_channels'].astype(np.int32).
      values

# Train CatBoost model on full training data
num_classes = int(y_train.max() + 1)
vc = pd.Series(y_train).value_counts()
class_weights = {int(k): float(len(y_train) / (
      num_classes * v)) for k, v in vc.items()}

train_pool = Pool(train_fe[feature_cols], y_train,
      cat_features=cat_features)

model = CatBoostClassifier(
    loss_function='MultiClass',
    eval_metric='TotalF1:average=Macro',
    learning_rate=0.05,
    depth=8,
    iterations=1200,
    random_state=42,
    class_weights=class_weights,
    verbose=200
)
model.fit(train_pool)

# Build features for test and predict
test_fe = make_features(test.copy())
X_test = test_fe[feature_cols]
test_pool = Pool(X_test, cat_features=cat_features)
test_preds = model.predict(test_pool).astype(int).
      flatten()

# Create submission using test.csv time ids
submission = pd.DataFrame({
    'time': test['time'].values,
    'open_channels': test_preds.astype(int)
})
submission.to_csv(os.path.join(final_dir, 'submission.
      csv'), index=False)
```

J KB Unit Example

This section shows one knowledge-base unit example stored in our curated KB.

KB Unit Example (JSON Format)


```
{
  "id": "f5f288c3c299e65a",
  "meta_tags": [
    "windowing",
    "aggregation",
    "segment_features"
  ],
  "taxonomy": {
    "modality": "Time_Series",
    "task_type": "Classification",
    "challenge": "Feature_Engineering",
```

```
    "solution_family": "Aggregation",
    "path": "Time_Series/Classification/
        Feature_Engineering/Aggregation"
  },
  "preview_card": {
    "method_name": "Windowing + aggregated
         statistics (sliding or block)",
    "problem_type": "Long sequences where
        labels/targets correspond to
        segments or depend on local
        context rather than the entire
        series",
    "applicability": "Effective when
        signals exhibit regime changes
        and local patterns; useful as a
        baseline feature set for tree/
        linear models or as auxiliary
        features for neural nets",
    "risk_warning": "If windows cross
        natural boundaries, leakage or
        label smearing can occur; window
        size/stride are sensitive
        hyperparameters and should be
        tuned via validation",
    "expected_improvement": "2-10%
        relative improvement versus using
         only point-wise values in models
         that cannot natively model
        sequences",
    "conditions": {
      "max_samples": 5000000,
      "min_samples": 100,
      "requires_gpu": false,
      "estimated_time_factor": 1.2,
      "compatible_models": ["xgboost", "
          lightgbm"],
      "requires_numerical": true,
      "handles_missing": true,
      "description": "Use when you need
          fixed-size feature vectors from
           sequences; include robust
          stats (median, MAD), energy/abs-
          mean, quantiles, and optionally
           features from denoised and raw
           signals."
    },
    "metric_impact": {
      "primary_metrics": ["auc", "f1"],
      "trade_off_metrics": ["latency"],
      "trade_off_description": "More
          features/windows increase
          compute and memory"
    },
    "code_stages": [
      "feature_engineering"
    ],
    "quick_tags": [
      "baseline_strong",
      "scalable",
      "model_agnostic"
    ]
  },
  "implementation": {
    "dependencies": [
      "numpy",
      "pandas",
      "scikit-learn"
    ],
    "usage_example": "X_win, y_win =
        window_aggregate_features(
        signal_array, labels=y_array,
        window_length=500, step_size=500,
         label_alignment='last')",
    "strategy_description": "Transform a
        long, high-frequency time series
        into a supervised learning table
        by splitting it into fixed-length
         windows and computing summary
        statistics per window.",
    "integration_notes": "Use during
        feature_engineering after basic
        cleaning/denoising and before
        model training. If data contains
        multiple independent sequences,
        process each sequence separately
        to avoid crossing boundaries.",
    "pseudocode": [
      "1. Choose window_length, step_size,
           and label_alignment.",
      "2. Process each sequence separately
           so windows do not cross
          boundaries.",
      "3. Compute aggregate features for
          each window.",
      "4. Train a classifier/regressor on
          the resulting table.",
      "5. If needed, map window
          predictions back to sample-
          level outputs."
    ],
    "key_parameters": {
      "window_length": "Controls temporal
          context versus sample count.",
      "step_size": "Controls overlap and
          compute cost.",
      "boundary_handling": "Usually forbid
           crossing sequence/session
          boundaries.",
      "time_aware_cv": "Use blocked or
          group-aware validation to avoid
           leakage."
    }
  },
  "source": {
    "notebook_title": "Ion Switching
        Competition : Signal EDA",
    "competition": "University of
        Liverpool - Ion Switching",
    "medal": "bronze"
  },
  "solves_challenges": [
    "Feature_Engineering",
    "Non_Stationarity"
  ],
  "generalizability_score": 5,
  "version": "3.0"
}
```