CAN LANGUAGE MODELS PROGRAM BETTER TIME-SERIES ANOMALY DETECTORS?

Anonymous authors
Paper under double-blind review

ABSTRACT

Language-model agents for time-series anomaly detection typically inspect series directly or orchestrate existing detectors to identify and explain anomalies. We introduce **AgentAD** (**Agent A**s **D**eveloper), in which the agent writes a standalone Python detector rather than serving as the detector itself. The resulting program runs without any language-model calls. We design an evidence-driven loop to help the agent develop detector programs that generalize to unseen series. The agent explores development series using code and submits candidate detector programs for evaluation. The framework evaluates each candidate on validation data, returning aggregate metrics and selected diagnostic cases through which the agent can inspect validation signals and labels. This bounded feedback is designed to help the agent diagnose failures, explore alternatives, and refine its code while reducing the risk of overfitting to the validation data. The framework retains the best-scoring detector program, while holdout series are reserved for final evaluation. We further train the agent with reinforcement learning in the same development environment, improving its ability to explore data, construct detector programs, and use diagnostic feedback to guide subsequent code revisions. Across multiple datasets and evaluation settings, detectors developed by AgentAD outperform established time-series anomaly detection methods. Under matched development budgets, reinforcement learning enables the trained agent to match or even surpass larger general-purpose language models at detector development. The agent framework, training pipeline, and model checkpoints are available at `https://anonymous.4open.science/status/AgentAD_anonymous-9D20`.

1 INTRODUCTION

Time-series anomaly detection (TSAD) depends on mechanisms suited to a signal's temporal structure, and their effectiveness varies across series (Liu & Paparrizos, 2024). A detector that captures isolated spikes may miss sustained shifts or mistake recurring peaks for anomalies. Developing a suitable detector therefore involves inspecting data, diagnosing failures, and revising preprocessing, features, fitting, or scoring. Language models can judge new inputs directly (Liu et al., 2025; Xu et al., 2026), while agents can assemble detectors into executable pipelines (Yang et al., 2025); executable anomaly rules can also be generated and refined (Gu et al., 2025). In contrast, this work adopts an **Agent as Developer** perspective, in which the agent iteratively develops a reusable fitting-and-scoring program that detects anomalies on unseen series without language-model calls.

AgentAD (Figure 1) organizes detector development as a sequence of evaluated code revisions. Feedback must expose concrete failures, yet repeated validation feedback can favor programs tailored to validation windows (Blum & Hardt, 2015). We address this tension through asymmetric data access: the agent examines train windows directly and receives aggregate metrics and bounded diagnostic cases from validation windows, while complete outer holdout series remain excluded from development. Because feedback guides both revision and candidate selection, development scores alone cannot establish transfer. We evaluate under *Segment Holdout*, which withholds windows within a series, and *Series Holdout*, which reserves complete series for final evaluation.

Detector development also requires learning across successive experiments. An experiment may reveal a useful failure mode without immediately improving the retained detector, so a terminal reward provides coarse supervision for intermediate actions. Building on DAPO (Yu et al., 2025), we

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

introduce *commit-level DAPO* (**C-DAPO**), combining final detector quality with a small auxiliary signal based on later improvements in selection quality. We compare developed detectors with established TSAD methods and compare the trained developer with the SFT-initialized policy and trajectory-level DAPO under matched development budgets.

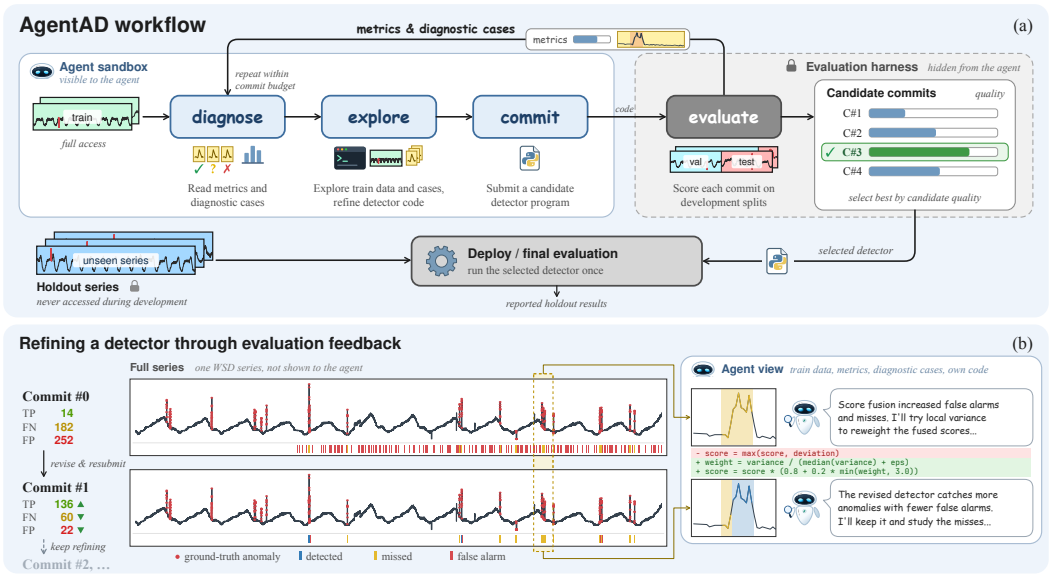

Figure 1: **AgentAD overview.** (a) The agent develops a reusable detector through evaluation feedback; the selected program detects anomalies in unseen series without further language-model calls. (b) A WSD example illustrating feedback-guided revision. The full series is not visible to the agent, which sees only the training data and selected diagnostic cases, together with aggregate metrics.

2 Related Work

2.1 Deep Learning Methods

Neural TSAD methods typically learn representations or fit parameters within detector designs specified in advance. MOMENT and TSPulse learn general time-series representations, while DADA and Time-RCD focus on transfer for anomaly detection (Goswami et al., 2024; Ekambaram et al., 2026; Shentu et al., 2025; Lan et al., 2026). Patch-based methods use local representations: PaAno compares embeddings with normal patches, CARLA and PatchAD learn contrastive representations, and ViT4TS scores visual features from rendered windows (Park & Kang, 2026; Darban et al., 2025; Zhong et al., 2025; He et al., 2026). Other methods, such as KAN-AD, Anomaly Transformer, and TranAD, fit model parameters on target or related series within fixed architectures and training objectives (Zhou et al., 2025; Xu et al., 2022; Tuli et al., 2022). AgentAD can reuse these components while revising preprocessing, fitting, and scoring in code.

2.2 Language Models and Agent Methods

Language models can participate directly in anomaly detection from numerical or visual inputs. LLMAD and ViTs make anomaly decisions with language or vision-language models, while VLM4TS uses a VLM to refine candidate intervals (Liu et al., 2025; Wang et al., 2026; He et al., 2026). In these methods, model inference remains part of detection for new inputs.

Agents can also coordinate existing tools and analyzers. AnomaMind gathers evidence from multiple tools for anomaly decisions, TSAD-Agents separates planning and checking roles, SAGE combines numerical and visual analyzers, and AD-AGENT selects libraries and builds executable pipelines (Tao et al., 2026; Xu et al., 2026; Kang et al., 2026; Yang et al., 2025). These systems

automate the use and composition of existing components, but their behavior remains bounded by the tools and mechanisms available to the agent.

Program-generating methods are closest to our setting. Argos generates and refines executable anomaly-detection rules, while related systems learn symbolic rules or generate feature functions within fixed scoring pipelines (Gu et al., 2025; Zhang & Jain, 2026; Hili et al., 2026). AgentAD goes beyond modifying individual rules or features by allowing revisions to preprocessing, fitting, and scoring logic across the detector program. In code synthesis, RLEF trains models to use execution feedback for iterative refinement (Gehring et al., 2025), while AgentAD trains the agent using detector quality and commit-level gains from the development process.

2.3 Benchmark and Metrics

TSB-AD contains 1,070 time series from 40 datasets and provides a standardized benchmark for detector comparison (Liu & Paparrizos, 2024). We use VUS-PR as the primary metric and report VUS-ROC, AUC-ROC, and Range-F1; the VUS metrics integrate range-aware ROC or PR performance over detection thresholds and temporal tolerance widths (Paparrizos et al., 2022). Appendix A specifies corpus filtering, holdout protocols, and metric aggregation.

3 AgentAD

AgentAD develops detector programs through evidence-driven revision and asymmetric data access (Figure 1).

3.1 Detector Development as a Programming Task

Given a collection of related time series, detector development is the task of constructing a reusable anomaly detection program (Appendix B.1). Available anomaly labels can help tailor the detector to the characteristics of the data, and the resulting program can be applied to held-out series (Appendix A.3).

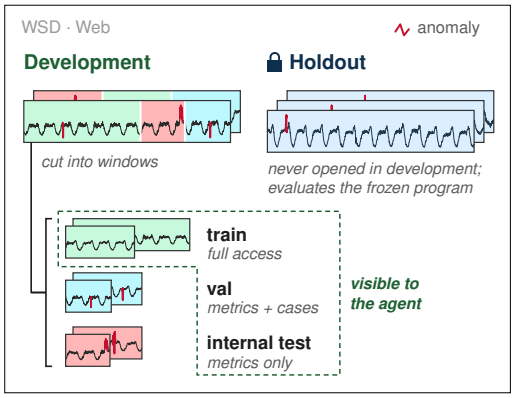

Figure 2: **Data partitioning in Series Holdout.** Development series are divided into train, validation, and internal-test windows with asymmetric access, while complete holdout series are reserved for final evaluation.

3.2 Evidence-Driven Iterative Development

In AgentAD, the agent is tasked with writing detector code in a sandbox equipped with Python and auxiliary tools (Figure 1). It can explore training series, experiment with detection approaches, and iteratively revise and submit candidate programs within a limited development budget. An evaluation harness executes each submitted program and returns aggregate performance metrics and diagnostic cases. The agent can inspect these cases to assess the detector's suitability for the data, adjust its detection logic, and submit the revised program for further evaluation. Appendices B.1–B.3 detail the development environment, feedback, and iteration process.

3.3 Data Partitioning and Detector Selection

We evaluate AgentAD in two settings, Segment Holdout and Series Holdout, with Series Holdout as the primary setting for our reported results (Appendix A.3). Under this setting, we group related series into classes and randomly split each class into development and holdout sets. The development series support detector construction, while the holdout series are reserved for evaluating the final detector. Each development series is further divided into train, validation, and internal-test windows with asymmetric access (Figure 2; Appendices A.2 and A.3).

Table 1: **Detection performance on TSB-AD.** Ser. and Seg. denote Series and Segment Holdout. AgentAD rows report the performance of detectors developed by the named language models. Scores are averaged within and then across classes.

Method	VUS-PR		VUS-ROC		AUC-ROC		Range-F1	
	Ser.	Seg.	Ser.	Seg.	Ser.	Seg.	Ser.	Seg.
Classical methods								
K-Means Distance	0.3698	0.3961	0.7417	0.7312	0.7219	0.7115	0.1855	0.1965
Spectral Residual	0.3366	0.3586	0.7355	0.7316	0.6877	0.6823	0.1660	0.1721
One-Class SVM	0.2866	0.3118	0.7062	0.7072	0.6392	0.6399	0.1331	0.1418
PCA	0.2739	0.3032	0.6778	0.6905	0.6328	0.6434	0.1157	0.1211
Matrix Profile	0.2802	0.3024	0.6735	0.6703	0.6445	0.6404	0.1113	0.1174
Isolation Forest	0.2672	0.2975	0.7460	0.7549	0.6890	0.6966	0.1035	0.1111
Local Outlier Factor	0.1459	0.1714	0.6203	0.6222	0.5436	0.5443	0.1150	0.1264
Non-Transformer neural								
PaAno	0.2970	0.4686	0.6577	0.8169	0.6145	0.7909	0.1084	0.2266
TSPulse-FT	0.3883	0.4441	0.7700	0.8086	0.7331	0.7833	0.1846	0.2341
TSPulse-ZS	0.3806	0.4019	0.7244	0.7295	0.6806	0.6842	0.1778	0.1801
xLSTMAD	0.3337	0.3525	0.7459	0.7495	0.6948	0.6890	0.1482	0.1658
KAN-AD	0.3051	0.3523	0.7028	0.7321	0.6525	0.6780	0.1527	0.1716
DADA	0.3245	0.3482	0.7259	0.7284	0.6767	0.6788	0.1561	0.1654
ScatterAD	0.2856	0.3058	0.6699	0.6733	0.6064	0.6087	0.1364	0.1441
AERCA	0.2492	0.2605	0.6794	0.6812	0.6260	0.6253	0.1214	0.1358
CARLA	0.2095	0.2485	0.5728	0.5902	0.5762	0.5796	0.0869	0.0468
Transformer								
Time-RCD	0.4515	0.4783	0.8058	0.8096	0.7778	0.7818	0.1781	0.1868
Left	0.4335	0.4494	0.8009	0.7919	0.7633	0.7561	0.1927	0.1985
AxonAD	0.3607	0.4174	0.7624	0.7879	0.6996	0.7362	0.1501	0.1636
CrossAD	0.3434	0.3685	0.7353	0.7485	0.6834	0.6946	0.1514	0.1546
AgentAD								
Qwen3.5-4B	0.4659	0.5017	0.8184	0.8046	0.7829	0.7567	0.1654	0.1849
Qwen3.5-9B	0.4688	0.4872	0.8123	0.7919	0.7785	0.7406	0.1623	0.1863
Qwen3.5-35B-A3B	0.4851	0.5008	0.8161	0.8052	0.7849	0.7511	0.1820	0.1926
Qwen3.8-27B	0.5312	0.5891	0.8586	0.8443	0.8283	0.8021	0.2259	0.2024
DeepSeek-V4.1-Flash	0.6056	0.5903	**0.8958**	0.8469	**0.8786**	0.8024	0.2235	0.1970
GLM-5.3-Flash	0.5668	0.5716	0.8595	0.8337	0.8410	0.7927	0.2262	0.2201
GLM-5.3	**0.6297**	**0.6570**	0.8817	**0.8803**	0.8709	**0.8620**	**0.2493**	**0.2352**

Higher is better; best values are bold. Evaluation coverage is detailed in Appendix B.5. *Under Series Holdout, all methods are evaluated on the same complete holdout series. Under Segment Holdout, established methods use archive suffixes and AgentAD uses internal-test windows, with different fitting permissions (Appendix A.3).*

The agent can use Python to inspect the full training windows and their anomaly labels. After the agent submits a candidate program, the harness runs it on the validation and internal-test windows of the development series, returning metrics from both partitions but diagnostic cases containing signals and labels only from validation. This access policy is designed to provide sufficient evidence for detector development while limiting overfitting, as examined in our ablations (Section 5.1). At the end of development, the framework selects the candidate with the highest mean validation and internal-test quality Q (Appendix A.4) and applies the same detector to all holdout series.

3.4 Detection Performance

Evaluation setup. We compare established TSAD methods with detectors developed by different language-model agents on TSB-AD (Liu & Paparrizos, 2024) (Appendix A.1). We use VUS-PR (Paparrizos et al., 2022) as the primary metric and report VUS-ROC, AUC-ROC, and Range-F1. Metric definitions and aggregation are in Appendix A.4, and developer configurations in Appendix B.4.

Main results. Under Series Holdout, AgentAD achieves higher mean VUS-PR than all established methods in Table 1 across all seven developer models. Even Qwen3.5-4B develops detectors

that surpass the strongest established method, Time-RCD (0.4659 versus 0.4515), while GLM-5.3 reaches 0.6297. AgentAD achieves the highest VUS-ROC, AUC-ROC, and Range-F1, with different developers leading on different metrics. The advantage is less consistent for Range-F1, which is sensitive to the choice of decision threshold, a known limitation discussed by Paparrizos et al. (2022). Under Segment Holdout, all seven developers exceed the established methods in reported mean VUS-PR. Evaluation windows and fitting permissions differ between the two groups in this setting, as noted in Table 1.

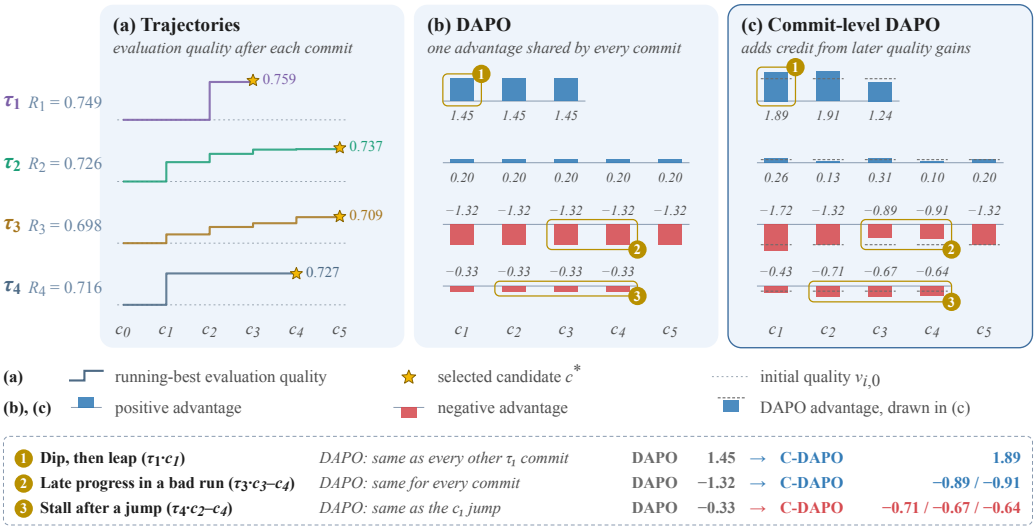

Figure 3: **Commit-level credit assignment.** For the development trajectories in (a), C-DAPO (c) refines the trajectory-level advantages of DAPO (b) using subsequent gains in selection quality at each commit. These gains capture intermediate development progress that the final reward alone does not distinguish, providing finer-grained supervision while keeping final detector quality as the primary objective. Appendix C.3 gives the calculations.

4 Reinforcement Learning

We train the agent to improve detectors through successive experiments. Iterative development requires an objective and feedback that guides further revisions. A *commit* marks a point at which the agent submits its current solution and receives feedback on progress toward that objective. In AgentAD, each commit evaluates a candidate detector against a configured quality objective and returns performance metrics and diagnostic feedback (Appendices A.4 and B.2). Optimizing final detector quality alone does not ensure that the agent learns to improve through iteration. SCoRe finds that multi-turn RL can favor producing a strong initial answer followed by little substantive revision, instead of learning to correct it (Kumar et al., 2025). Such behavior can limit exploration during detector development.

We introduce *commit-level DAPO* (**C-DAPO**) to train the agent for iterative detector development (Figure 3). Building on DAPO, we use the selection quality returned after each commit to refine the trajectory-level advantage, constructing finer-grained advantage signals that reflect progress during development.

4.1 Commit-Level Credit Assignment

Each commit closes a development segment. For trajectory i, let $v_{i,t-1}$ be the best selection quality attained before segment t and v_i^{final} the best quality at termination, following AgentAD's selection

rule (Section 3.3). We define the subsequent gain as:

$$g_{i,t} = v_i^{\text{final}} - v_{i,t-1}. \tag{1}$$

Using the historical best avoids counting a return to a previously attained score as new progress. Including later improvements allows exploration to receive credit even when its immediate outcome does not improve the detector.

We compare these gains across eligible trajectories for the same task at the same commit position. This normalization, denoted by Norm_t, produces a signed auxiliary advantage that adjusts the trajectory advantage A_i^{traj}, obtained by normalizing terminal rewards across trajectories:

$$\widetilde{A}_{i,t} = A_i^{\text{traj}} + \alpha \underbrace{\mathrm{Norm}_t(g_{i,t})}_{A_{i,t}^{\text{com}}}. \tag{2}$$

The trajectory term preserves the preference for high final quality, while the auxiliary term adjusts the update strength for each development segment. We use a small auxiliary weight α and prevent the auxiliary term from reversing the trajectory advantage's direction. The resulting advantages enter DAPO's token-level objective (Yu et al., 2025). Appendix C.3 defines the rewards, normalization, and direction protection; Appendix C.4 gives the sampling procedure and full optimization objective.

Table 2: **Detection performance after developer-policy training.** Ser. and Seg. denote Series and Segment Holdout on the same seven classes excluded from training. Rows report the performance of detectors developed by the named policies, averaged over series within each class and then across classes.

Developer policy	VUS-PR		VUS-ROC		AUC-ROC		Range-F1	
	Ser.	Seg.	Ser.	Seg.	Ser.	Seg.	Ser.	Seg.
Qwen3.5-35B-A3B-DAPO	0.5209	0.5234	0.8441	0.8874	0.8054	0.8536	0.2005	0.2340
Qwen3.5-35B-A3B-SFT	0.5099	0.5002	0.8424	0.8558	0.8022	0.8217	0.1761	0.2325
Qwen3.5-35B-A3B-C-DAPO	0.5496	0.5412	**0.8504**	0.8918	**0.8235**	0.8720	0.2553	0.2867
Qwen3.5-35B-A3B	0.4949	0.4609	0.8206	0.8032	0.7808	0.7767	0.1540	0.2101
Qwen3.8-27B	0.5526	0.5210	0.8192	0.8482	0.7939	0.8266	0.3280	0.2687
DeepSeek-V4.1-Flash	0.5364	0.5639	0.8057	0.8904	0.7838	0.8616	0.2997	0.2177
GLM-5.3-Flash	0.5965	**0.6109**	0.8172	**0.9316**	0.7919	**0.9107**	0.2764	**0.3312**
GLM-5.3	**0.6568**	0.5553	0.8394	0.8917	0.8110	0.8709	**0.3323**	0.2849

Higher is better; best values are bold. DAPO and C-DAPO use their final-epoch checkpoints. All policies use the main comparison's AgentAD evaluation settings. Evaluation protocols and settings are detailed in Appendix C.5.

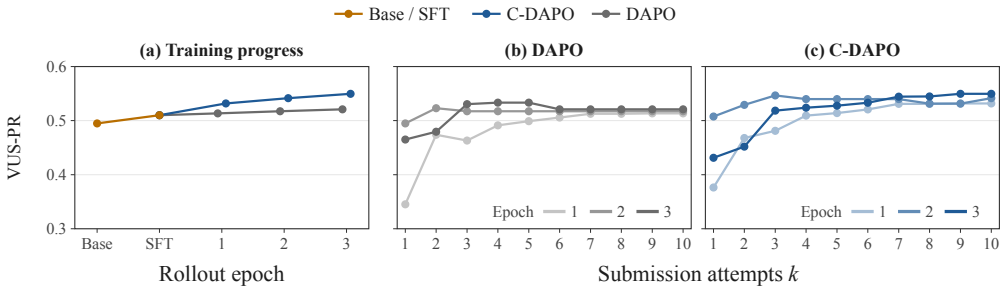

Figure 4: **Policy training and detector development.** All panels report mean VUS-PR on the seven Series Holdout classes used in Table 2. (a) Final detector performance for the base model, the SFT policy, and DAPO and C-DAPO checkpoints after each rollout epoch. (b–c) Detector performance over successive submission attempts. Each point evaluates the detector selected at that stage, and darker curves correspond to later training epochs. These classes are excluded from training (Appendix C.1); holdout scores are used only for evaluation and do not guide detector selection or advantage construction (Appendix C.5).

4.2 Training and Results

We fine-tune Qwen3.5-35B-A3B on detector-development trajectories collected with GLM-5.3-Flash. DAPO and C-DAPO are then trained from this shared SFT checkpoint using the same settings, with C-DAPO adding the auxiliary advantage described in Section 4.1. Training tasks and SFT initialization are described in Appendices C.1–C.2; RL settings and evaluation protocols are given in Appendices C.4–C.5.

C-DAPO improves detector performance over SFT and DAPO across all four metrics in both holdout settings (Table 2). Under Series Holdout, the trained 35B developer also outperforms DeepSeek-V4.1-Flash (552B) in VUS-PR. The gap between C-DAPO and DAPO widens during training (Figure 4(a)). At the final checkpoint, C-DAPO starts with lower first-attempt performance than DAPO but continues to improve through later submissions, producing a better final detector (Figure 4(b–c)). Section 5.4 examines these differences in development behavior.

5 Ablation and Analysis

We examine how iterative development, diagnostic feedback, and data access affect detector quality, and how policy training changes development behavior. Framework ablations use Qwen3.8-27B on seven Series Holdout classes; the policy-training analysis compares Qwen3.5-35B-A3B variants on these classes. Detailed settings and complete results are provided in Appendices B.6, B.7, and C.5.

5.1 Framework Ablations

Full AgentAD outperforms First Commit and Python-only Development (Table 3), supporting repeated evaluation and revision. Removing diagnostic cases weakens results despite continued iteration: aggregate scores offer less guidance than examples of success and failure. Holdout performance declines from Full AgentAD to No Test Feedback and further to Full Validation Access, consistent with greater susceptibility to overfitting as development relies more on validation data. These observations support evidence-driven iteration with asymmetric access; Section 5.2 examines how this affects development and selection.

Table 3: **Framework ablations.**

Setting	VUS-PR	Range-F1
Full AgentAD	**0.5864**	**0.3416**
First Commit	0.3942	0.2593
Python-only Development	0.3870	0.3145
No Diagnostic Cases	0.4509	0.3138
No Test Feedback	0.5528	0.3118
Full Validation Access	0.5137	0.2576

5.2 Development Dynamics

Figure 5 compares the selected detector with the best candidate observed on holdout. Full AgentAD develops strong candidates and retains one close to the oracle. Without diagnostic cases, both Selected@k and Oracle Best@k remain low, suggesting limited progress in candidate development. The final development–holdout gap grows from Full AgentAD through No Test Feedback to Full Validation Access. With full validation access, strong candidates are found but not retained, despite high development scores. This divergence is consistent with overfitting to visible feedback and highlights the role of asymmetric access in candidate selection.

5.3 Development–Holdout Association

Table 4 reports within-task Pearson and Spearman correlations. No Diagnostic Cases has the strongest VUS-PR association but weaker detectors than Full AgentAD: tracking quality does not ensure useful feedback. No Test Feedback shows weak VUS-PR but strong Range-F1 association, indicating metric dependence. Definitions and task counts are in Appendix B.7.

Table 4: **Development–Holdout correlations.**

Setting	VUS-PR		Range-F1	
	r	ρ	r	ρ
Full AgentAD	0.668	0.507	0.758	0.789
No Test Feedback	0.062	0.202	0.769	0.823
Full Validation Access	0.238	0.247	0.431	0.433
No Diagnostic Cases	0.834	0.602	0.333	0.221

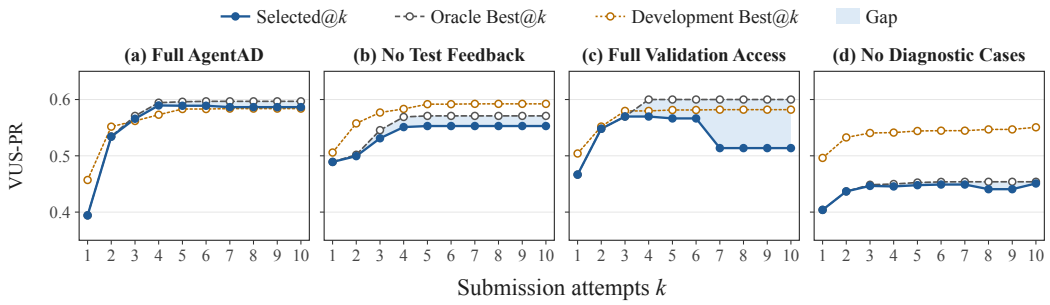

Figure 5: **Development and selection across commits.** VUS-PR averaged over the seven Series Holdout classes. Selected@k reports the selected detector's holdout performance after k submission attempts. Oracle Best@k reports the best holdout performance among candidates observed up to that point. Development Best@k reports the selected detector's performance on the development data, which reflects the agent's limited view of detector quality. The shaded gap shows how far the selected detector falls below the best candidate found. Selection rules and retrospective oracle calculations are detailed in Appendices B.6 and B.7. Holdout scores never guide development or selection.

5.4 Effect of Policy Training on Iteration

Table 5 separates first-commit quality from the holdout improvement achieved by final selection. DAPO produces the strongest first submission, but makes fewer commits and achieves a similar subsequent gain to SFT. This suggests that its improvement over SFT comes mainly from better initial detectors, consistent with the development curves in Figure 4. C-DAPO starts with a weaker detector but achieves the largest gain during development and the highest final quality. C-DAPO also achieves greater improvement than the untrained policy from a similar first-commit quality. Its higher commit and token counts indicate a longer development process, while the larger holdout improvement shows that this process yields better detectors. These results support stronger iterative development under C-DAPO, accompanied by higher development cost.

Table 5: **Detector improvement and development cost.**

Developer policy	First	Selected	Improvement	Commits	Tokens (M)
Qwen3.5-35B-A3B	0.3274	0.4949	0.1675	5.14	1.74
Qwen3.5-35B-A3B-DAPO	**0.3628**	0.5209	0.1581	4.43	2.19
Qwen3.5-35B-A3B-SFT	0.3515	0.5099	0.1585	5.57	2.60
Qwen3.5-35B-A3B-C-DAPO	0.3290	**0.5496**	**0.2206**	6.57	4.02

First and Selected report Series Holdout VUS-PR of the first scored submission and the final retained detector. Improvement is their paired difference (Selected minus First); the best scores and improvements are bold. All entries are means over the seven evaluation classes. DAPO and C-DAPO use their final-epoch checkpoints. Commits and tokens measure development cost per run; tokens are in millions. Appendix C.5 defines these comparisons and their shared AgentAD settings.

6 Conclusion

We developed AgentAD (Agent As Developer), a framework that automates time-series anomaly detector development through evidence-driven exploration and iterative code revision. Detectors developed by AgentAD outperform established methods in our experiments and operate on unseen series without language-model calls. We also developed C-DAPO (Commit-level DAPO), which uses progress across successive experiments to construct finer-grained advantages for training the developer agent. C-DAPO improves the agent's iterative development ability, leading to larger gains during refinement and stronger final detectors than SFT and trajectory-level DAPO. Under matched development budgets, the trained agent can match or even surpass larger general-purpose language models at detector development.

AI USE STATEMENT

Language models are the central subject of this research. We study their ability to develop and revise standalone anomaly-detector programs in AgentAD and train their development behavior, as described in Sections 3 and 4. Separately, generative AI assistants helped implement and revise parts of the research code, including data-partitioning and framework components; they also helped polish the language of this manuscript and prepare figures. The authors reviewed the AI-assisted code, text, and figures and take responsibility for the final paper and artifacts.

ETHICS STATEMENT

AgentAD develops executable anomaly detectors from existing time-series benchmarks. A detector's benchmark score does not establish that it is safe to deploy: false alarms and missed events may carry different costs across applications, and data distributions may change after deployment. Any operational use requires application-specific validation and human oversight. Agent-generated programs should be inspected and run with appropriate isolation and resource limits, particularly when they process sensitive data or interact with production systems.

REPRODUCIBILITY STATEMENT

Appendix A describes the benchmark corpus, data partitions, information access, evaluation metrics, aggregation, and candidate selection. Appendix B details detector evaluation, diagnostic feedback, context management, and development settings. Appendix C specifies the training tasks, SFT initialization, reinforcement learning objective and optimization, and developer-policy evaluation. Appendix D provides data visualizations and complete detector programs. The anonymous repository provides the implementation and configuration details needed to reproduce the experiments.

REFERENCES

Avrim Blum and Moritz Hardt. The ladder: A reliable leaderboard for machine learning competitions. In Francis Bach and David Blei (eds.), *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pp. 1006–1014, Lille, France, 07–09 Jul 2015. PMLR. URL https://proceedings.mlr.press/v37/blum15.html.

Zahra Zamanzadeh Darban, Geoffrey I. Webb, Shirui Pan, Charu C. Aggarwal, and Mahsa Salehi. Carla: Self-supervised contrastive representation learning for time series anomaly detection. *Pattern Recognition*, 157:110874, 2025. ISSN 0031-3203. doi: https://doi.org/10.1016/j.patcog.2024.110874. URL https://www.sciencedirect.com/science/article/pii/S0031320324006253.

Vijay Ekambaram, Subodh Kumar, Arindam Jati, Sumanta Mukherjee, Tomoya Sakai, Pankaj Dayama, Wesley Gifford, and Jayant Kalagnanam. Tspulse: Tiny pre-trained models with disentangled representations for rapid time-series analysis. In C. Vondrick, B. Hariharan, C. Raffel, L. Pinto, D. Yang, and A. Faust (eds.), *International Conference on Learning Representations*, volume 2026, pp. 62090–62133, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/file/6572721b15834cb37599abec23b5784f-Paper-Conference.pdf.

Jonas Gehring, Kunhao Zheng, Jade Copet, Vegard Mella, Taco Cohen, and Gabriel Synnaeve. RLEF: Grounding code LLMs in execution feedback with reinforcement learning. In Aarti Singh, Maryam Fazel, Daniel Hsu, Simon Lacoste-Julien, Felix Berkenkamp, Tegan Maharaj, Kiri Wagstaff, and Jerry Zhu (eds.), *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 19034–19055. PMLR, 13–19 Jul 2025. URL https://proceedings.mlr.press/v267/gehring25a.html.

Mononito Goswami, Konrad Szafer, Arjun Choudhry, Yifu Cai, Shuo Li, and Artur Dubrawski. MOMENT: A family of open time-series foundation models. In Ruslan Salakhutdinov, Zico

Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp (eds.), *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 16115–16152. PMLR, 21–27 Jul 2024. URL https://proceedings.mlr.press/v235/goswami24a.html.

Yile Gu, Yifan Xiong, Jonathan Mace, Yuting Jiang, Yigong Hu, Baris Kasikci, and Peng Cheng. Argos: Agentic time-series anomaly detection with autonomous rule generation via large language models, 2025. URL https://arxiv.org/abs/2501.14170.

Zelin He, Sarah Alnegheimish, and Matthew Reimherr. Harnessing vision-language models for time series anomaly detection. In *Proceedings of the Fortieth AAAI Conference on Artificial Intelligence and Thirty-Eighth Conference on Innovative Applications of Artificial Intelligence and Sixteenth Symposium on Educational Advances in Artificial Intelligence*, AAAI'26/IAAI'26/EAAI'26. AAAI Press, 2026. ISBN 978-1-57735-906-7. doi: 10.1609/aaai.v40i26.39319. URL https://doi.org/10.1609/aaai.v40i26.39319.

Youssef Attia El Hili, Malik Tiomoko, and Corinne Ancourt. Llm-generated feature pools for time series anomaly detection, 2026. URL https://arxiv.org/abs/2609.21801.

Hyeongwon Kang, Jeongseob Kim, Jinwoo Park, and Pilsung Kang. Detecting time series anomalies like an expert: A multi-agent llm framework with specialized analyzers, 2026. URL https://arxiv.org/abs/2605.05725.

Aviral Kumar, Vincent Zhuang, Rishabh Agarwal, Yi Su, JD Co-Reyes, Avi Singh, Kate Baumli, Shariq Iqbal, Colton Bishop, Rebecca Roelofs, Lei Zhang, Kay McKinney, Disha Shrivastava, Cosmin Paduraru, George Tucker, Doina Precup, Feryal Behbahani, and Aleksandra Faust. Training language models to self-correct via reinforcement learning. In Y. Yue, A. Garg, N. Peng, F. Sha, and R. Yu (eds.), *International Conference on Learning Representations*, volume 2025, pp. 54523–54549, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/871ac99fdc5282d0301934d23945ebaa-Paper-Conference.pdf.

Tian Lan, Hao Duong Le, Jinbo Li, Wenjun He, Meng Wang, Chenghao Liu, and Chen Zhang. Towards foundation models for zero-shot time series anomaly detection: Leveraging synthetic data and relative context discrepancy. In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=yXqnyIvGAy.

Jun Liu, Chaoyun Zhang, Jiaxu Qian, Minghua Ma, Si Qin, Chetan Bansal, Qingwei Lin, Saravan Rajmohan, and Dongmei Zhang. Large language models can deliver accurate and interpretable time series anomaly detection. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V.2*, KDD '25, pp. 4623–4634, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400714542. doi: 10.1145/3711896.3737239. URL https://doi.org/10.1145/3711896.3737239.

Qinghua Liu and John Paparrizos. The elephant in the room: Towards a reliable time-series anomaly detection benchmark. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (eds.), *Advances in Neural Information Processing Systems*, volume 37, pp. 108231–108261. Curran Associates, Inc., 2024. doi: 10.52202/079017-3437. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/c3f3c690b7a99fba16d0efd35cb83b2c-Paper-Datasets_and_Benchmarks_Track.pdf.

John Paparrizos, Paul Boniol, Themis Palpanas, Ruey S. Tsay, Aaron Elmore, and Michael J. Franklin. Volume under the surface: a new accuracy evaluation measure for time-series anomaly detection. *Proc. VLDB Endow.*, 15(11):2774–2787, July 2022. ISSN 2150-8097. doi: 10.14778/3551793.3551830. URL https://doi.org/10.14778/3551793.3551830.

Jinju Park and Seokho Kang. Paano: Patch-based representation learning for time-series anomaly detection, 2026. URL https://arxiv.org/abs/2602.01359.

Qichao Shentu, Beibu Li, Kai Zhao, Yang Shu, Zhongwen Rao, Lujia Pan, Bin Yang, and Chenjuan Guo. Towards a general time series anomaly detector with adaptive bottlenecks and dual adversarial decoders. In Y. Yue, A. Garg, N. Peng, F. Sha, and R. Yu

(eds.), *International Conference on Learning Representations*, volume 2025, pp. 81358–81381, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/ca7998666c2e53cc1e882b7268414d8a-Paper-Conference.pdf.

Xiaoyu Tao, Yuchong Wu, Mingyue Cheng, Ze Guo, and Tian Gao. Anomamind: Agentic time series anomaly detection with tool-augmented reasoning, 2026. URL https://arxiv.org/abs/2602.13807.

Shreshth Tuli, Giuliano Casale, and Nicholas R. Jennings. Tranad: deep transformer networks for anomaly detection in multivariate time series data. *Proc. VLDB Endow.*, 15(6):1201–1214, February 2022. ISSN 2150-8097. doi: 10.14778/3514061.3514067. URL https://doi.org/10.14778/3514061.3514067.

Zexin Wang, Changhua Pei, Yang Liu, Hengyue Jiang, Quan Zhou, Haotian Si, Hang Cui, Jianhui Li, Gaogang Xie, Jingjing Li, and Dan Pei. Vits: Teaching machines to see time series anomalies like human experts. In *Proceedings of the ACM Web Conference 2026*, WWW '26, pp. 5286–5297, New York, NY, USA, 2026. Association for Computing Machinery. ISBN 9798400723070. doi: 10.1145/3774904.3792323. URL https://doi.org/10.1145/3774904.3792323.

Jiehui Xu, Haixu Wu, Jianmin Wang, and Mingsheng Long. Anomaly transformer: Time series anomaly detection with association discrepancy. In *International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=LzQQ89U1qm_.

Xiongxiao Xu, Haoran Wang, Yueqing Liang, Philip S. Yu, Yue Zhao, and Kai Shu. Can multimodal llms perform time series anomaly detection? In *Proceedings of the ACM Web Conference 2026*, WWW '26, pp. 5392–5403, New York, NY, USA, 2026. Association for Computing Machinery. ISBN 9798400723070. doi: 10.1145/3774904.3792376. URL https://doi.org/10.1145/3774904.3792376.

Tiankai Yang, Junjun Liu, Michael Siu, Jiahang Wang, Zhuangzhuang Qian, Chanjuan Song, Cheng Cheng, Xiyang Hu, and Yue Zhao. AD-AGENT: A multi-agent framework for end-to-end anomaly detection. In Kentaro Inui, Sakriani Sakti, Haofen Wang, Derek F. Wong, Pushpak Bhattacharyya, Biplab Banerjee, Asif Ekbal, Tanmoy Chakraborty, and Dhirendra Pratap Singh (eds.), *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*, pp. 191–205, Mumbai, India, December 2025. The Asian Federation of Natural Language Processing and The Association for Computational Linguistics. ISBN 979-8-89176-303-6. doi: 10.18653/v1/2025.findings-ijcnlp.11. URL https://aclanthology.org/2025.findings-ijcnlp.11/.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, juncai liu, LingJun Liu, Xin Liu, Haibin Lin, Zhiqi Lin, Bole Ma, Guangming Sheng, Yuxuan Tong, Chi Zhang, Mofan Zhang, Ru Zhang, Wang Zhang, Hang Zhu, Jinhua Zhu, Jiaze Chen, Jiangjie Chen, Chengyi Wang, Hongli Yu, Yuxuan Song, Xiangpeng Wei, Hao Zhou, Jingjing Liu, Wei-Ying Ma, Ya-Qin Zhang, Lin Yan, Yonghui Wu, and Mingxuan Wang. Dapo: An open-source llm reinforcement learning system at scale. In D. Belgrave, C. Zhang, H. Lin, R. Pascanu, P. Koniusz, M. Ghassemi, and N. Chen (eds.), *Advances in Neural Information Processing Systems*, volume 38, Main Conference, pp. 113222–113244. Curran Associates, Inc., 2025. doi: 10.52202/085713-3775. URL https://proceedings.neurips.cc/paper_files/paper/2025/file/a4277440d50f1f15d2cb4c14f7e0c0d2-Paper-Conference.pdf.

Haoting Zhang and Shekhar Jain. Llm-assisted logic rule learning: Scaling human expertise for time series anomaly detection, 2026. URL https://arxiv.org/abs/2601.19255.

Zhijie Zhong, Zhiwen Yu, Yiyuan Yang, Weizheng Wang, Kaixiang Yang, and C. L. Philip Chen. Patchad: A lightweight patch-based mlp-mixer for time series anomaly detection. *IEEE Transactions on Big Data*, 11(6):3460–3473, 2025. doi: 10.1109/TBDATA.2025.3596745.

Quan Zhou, Changhua Pei, Fei Sun, Han Jing, Zhengwei Gao, Haiming Zhang, Gaogang Xie, Dan Pei, and Jianhui Li. KAN-AD: Time series anomaly detection with kolmogorov–arnold networks.

In Aarti Singh, Maryam Fazel, Daniel Hsu, Simon Lacoste-Julien, Felix Berkenkamp, Tegan Maharaj, Kiri Wagstaff, and Jerry Zhu (eds.), *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 79136–79149. PMLR, 13–19 Jul 2025. URL https://proceedings.mlr.press/v267/zhou25u.html.

APPENDIX

A DATA AND EVALUATION PROTOCOL

This appendix describes the benchmark corpus, task construction, data partitions, and evaluation protocols, including information access and metric aggregation. Full implementation details and hyperparameters are provided in the anonymous repository.

A.1 CORPUS AND TASK GROUPS

We use TSB-AD as the benchmark corpus (Liu & Paparrizos, 2024). It contains 1,070 time series from multiple public datasets: 870 univariate and 200 multivariate. Among the univariate series, 170 are extracted from multivariate sources, with the entity-level anomaly label assigned to every channel. An entity-level event need not be anomalous in every channel, so we do not treat these broadcast labels as reliable channel-level ground truth. We exclude the 170 derived series while retaining the original multivariate series. The excluded series come from CATSv2, Daphnet, Exathlon, LTDB, MITDB, MSL, OPPORTUNITY, SMAP, SMD, SVDB, SWaT, and TAO. This filtering leaves 900 series: 700 univariate and 200 multivariate.

We group the retained series by source dataset, domain, and channel schema, yielding 49 classes. Each class defines a common detector-development task for the compared methods. Univariate and multivariate series from the same source remain separate, as do series with distinct channel schemas, even when they have the same number of channels. Class names use the dataset name as the base identifier, a numeric suffix to distinguish channel schemas within a dataset when needed, and a domain abbreviation after a middle dot. For example, `NAB·Env` and `NAB·Fac` differ by domain. Table 6 summarizes the 49 classes by source dataset, domain, channel count, number of series, and mean series length.

Table 6: **Retained time-series classes.** The 49 classes group TSB-AD series by dataset, domain, and channel schema.

Class	Dataset	Domain	Channels	Series	Mean points
CATSv2·Sen	CATSv2	Sensor	17	6	240,000
CreditCard·Fin	CreditCard	Finance	29	1	284,807
Daphnet·Hum	Daphnet	Human activity	9	1	38,774
Exathlon-1·Fac	Exathlon	Facility	7	1	46,650
Exathlon-2·Fac	Exathlon	Facility	8	1	46,656
Exathlon-3·Fac	Exathlon	Facility	12	1	46,649
Exathlon-4·Fac	Exathlon	Facility	12	2	46,655
Exathlon-5·Fac	Exathlon	Facility	16	1	46,660
Exathlon-6·Fac	Exathlon	Facility	18	8	46,211
Exathlon-7·Fac	Exathlon	Facility	19	5	129,197
Exathlon-8·Fac	Exathlon	Facility	31	8	43,515
GECCO·Sen	GECCO	Sensor	9	1	138,521
Genesis·Sen	Genesis	Sensor	18	1	16,220
GHL·Sen	GHL	Sensor	19	25	199,001
IOPS·Web	IOPS	Web service	1	17	72,792
LTDB-1·Med	LTDB	Medical	2	4	100,000
LTDB-2·Med	LTDB	Medical	3	1	100,000
MGAB·Syn	MGAB	Synthetic	1	9	97,778
MITDB-1·Med	MITDB	Medical	2	8	346,250
MITDB-2·Med	MITDB	Medical	2	1	150,000
MITDB-3·Med	MITDB	Medical	2	3	316,667
MITDB-4·Med	MITDB	Medical	2	1	500,000
MSL·Sen	MSL	Sensor	55	16	3,119
NAB·Env	NAB	Environment	1	1	7,266
NAB·Fac	NAB	Facility	1	9	5,239
NAB·Syn	NAB	Synthetic	1	6	4,031
NAB·Tra	NAB	Traffic	1	6	3,725

Table 6, continued

Class	Dataset	Domain	Channels	Series	Mean points
NAB·Web	NAB	Web service	1	6	6,974
NEK·Web	NEK	Web service	1	9	1,073
OPPORTUNITY·Hum	OPPORTUNITY	Human activity	248	8	17,427
Power·Fac	Power	Facility	1	1	35,040
PSM·Fac	PSM	Facility	25	1	217,624
SED·Med	SED	Medical	1	3	23,332
SMAP·Sen	SMAP	Sensor	25	27	7,856
SMD·Fac	SMD	Facility	38	22	25,466
Stock·Fin	Stock	Finance	1	20	15,000
SVDB·Med	SVDB	Medical	2	31	207,123
SWaT-1·Sen	SWaT	Sensor	51	1	399,919
SWaT-2·Sen	SWaT	Sensor	66	1	14,996
TAO·Env	TAO	Environment	3	13	10,000
TODS·Syn	TODS	Synthetic	1	15	5,000
UCR·Env	UCR	Environment	1	16	11,275
UCR·Fac	UCR	Facility	1	63	138,726
UCR·Hum	UCR	Human activity	1	28	96,238
UCR·Med	UCR	Medical	1	107	34,985
UCR·Sen	UCR	Sensor	1	14	11,271
WSD·Web	WSD	Web service	1	111	17,509
YAHOO·Syn	YAHOO	Synthetic	1	92	1,421
YAHOO·Web	YAHOO	Web service	1	167	1,639

Mean points is the average number of time points per series.

A.2 Data Partitions

We use two levels of partitioning. An outer split divides the complete series within each class into development and holdout sets to evaluate transfer to unseen series. An inner split divides each development series into train, validation, and internal-test windows. All random data partitions use seed 42, including those used for SFT and RL. Appendix A.3 specifies how each evaluation protocol uses these partitions.

Random (by class). For the main Random (by class) split, we randomly divide the 900 retained series within each class into development and holdout sets at a target 7:3 ratio. For a class containing $n \geq 2$ series, we assign

$$n_{\text{hold}} = \min\{n - 1, \max(1, \lceil 0.3n \rceil)\}$$

series to holdout, ensuring that both sets remain nonempty. Singleton classes remain entirely in development. After rounding and singleton handling, the split contains 620 development series (483 univariate, 137 multivariate) and 280 holdout series (217 univariate, 63 multivariate). The 33 classes with holdout series form the Series Holdout evaluation set; the remaining 16 classes do not enter the reported average.

Independent evaluation and ablation subset. Framework ablations and independent developer-policy evaluation use the same seven univariate classes: `NAB·Web`, `NAB·Tra`, `NAB·Syn`, `TODS·Syn`, `SED·Med`, `UCR·Env`, and `IOPS·Web`. All series from these classes are excluded from SFT and RL training. Developer-policy evaluation uses both Series Holdout and Segment Holdout on these classes. Framework ablations use Series Holdout, with the same development and holdout splits across variants.

Development windows. Each development series with at least three points is divided into three nonempty, contiguous, half-open windows: train, validation, and internal test. They do not overlap, cover the complete timeline, and target a 1:1:1 length ratio. Because detection is offline, the train window need not precede the validation and internal-test windows. Shorter series are left unsplit. The resulting windows are treated as independent inputs, so temporal operations cannot cross partition boundaries.

Event allocation. The partitioner first allocates whole anomaly events to the three windows, then chooses cut locations that preserve the events and their surrounding normal context where possible. With $k \geq 3$ events, each window is assigned $\lfloor k/3 \rfloor$ events, and the remaining one or two go to internal test and then validation. With fewer events, the evaluation windows take priority: a single event is assigned to internal test, while two events are assigned one each to validation and internal test. In both cases, train is placed in the longest anomaly-free stretch and contains no labeled event; with a single event, validation contains none either. Normal-only series are partitioned by length.

Cuts are chosen to leave normal context around the assigned events. The preferred margin is 10% of the corresponding window length, reduced to 5% when necessary. If no feasible cut remains, the margin is relaxed further before the procedure falls back to a forced length-based split, which may divide an anomaly event.

Figure 10 shows the resulting windows and anomaly-event allocation across development series.

A.3 Evaluation Protocols and Information Access

Series Holdout evaluates transfer to complete series excluded from development and is our primary setting. Segment Holdout evaluates withheld portions of series in separate development runs. Both protocols use the Random (by class) split in Appendix A.2.

Series Holdout. All methods use the same classes and the same development and holdout splits. The 620 development series support detector construction; 604 of them belong to the 33 evaluated classes. AgentAD develops its detector using their train, validation, and internal-test windows defined in Appendix A.2. The 280 complete holdout series are reserved for final evaluation after development. They are excluded from exploration, feedback, fitting, configuration or program selection, and threshold calibration.

Segment Holdout. Segment Holdout uses the same 280 series selected as holdout series by the outer split, but defines development and evaluation portions within them. Established methods and AgentAD use different development and scoring procedures, described below, so their results provide a descriptive rather than strictly matched comparison.

Established methods under Segment Holdout. For each series, the benchmark provides an archive prefix ending at t_0, with $0 < t_0 < T$. The unlabeled prefix is available for detector preparation, and evaluation metrics are computed only on the suffix $t_0{:}T$. Suffix labels remain unavailable until predictions have been frozen.

AgentAD under Segment Holdout. AgentAD partitions each of these series into train, validation, and internal-test windows using the rules in Appendix A.2. The train and validation windows support detector development, and the internal-test windows provide the reported metrics. The 620 development series used by Series Holdout do not enter these separate runs.

AgentAD information access. In Full AgentAD, train signals and available labels are directly accessible to the agent for exploration; train windows are excluded from the reported evaluation metrics. The detector receives no labels when scoring. The evaluator uses the complete validation windows to compute metrics and calibrate a threshold, while the agent receives aggregate metrics and selected labeled diagnostic cases. Case disclosure is bounded per evaluation, as described in Appendix B.2.

The evaluator also scores the internal-test windows, but their signals, labels, pointwise scores, and diagnostic cases remain hidden from the agent in both protocols. Under Series Holdout, aggregate internal-test metrics are visible, and the selection score $S(c)$ (Equation 4) averages validation and internal-test quality. Under Segment Holdout, internal-test aggregates are also hidden, and $S(c)$ uses validation quality alone.

Table 7 summarizes Full AgentAD's information access and candidate selection under both protocols. Appendix B.6 specifies the changes made in the framework ablations. The quality objective Q is defined below, and policy-training access, selection, and reward rules are given in Appendix C.

Table 7: **Information access and data use in Full AgentAD.** Ser. and Seg. denote Series and Segment Holdout; entries without a prefix apply to both protocols.

	Visible to the agent			
Partition	**Signals**	**Labels**	**Metrics**	**Use**
Windows of each development series				
Train	✓	✓	✗	Local exploration and detector development
Validation	Cases	Cases	✓	Diagnostic feedback, threshold calibration, and selection
Internal test	✗	✗	Ser. only	Ser.: selection together with validation Seg.: final evaluation
Outer holdout under Series Holdout				
Holdout series	✗	✗	✗	Final evaluation

Development series are the 620 development series under Series Holdout and the 280 holdout-side series under Segment Holdout. *Cases* are selected validation slices with labels and predictions (Appendix B.2). Selection maximizes $S(c)$ (Equation 4).

A.4 Metrics, Objectives, and Aggregation

Reported metrics. We report VUS-PR, VUS-ROC, AUC-ROC, and Range-F1. VUS varies both the detection threshold and the temporal tolerance around each labeled anomaly range, forming precision-recall or ROC surfaces whose volumes are reported as VUS-PR and VUS-ROC. For both VUS metrics, we follow the implementation and computational details of Paparrizos et al. (2022). VUS-PR is our primary metric because anomalous points are sparse and precision-recall evaluation is more sensitive to false alarms. VUS-ROC and point-wise AUC-ROC provide complementary ranking measures, while Range-F1 measures interval-level detection performance.

Aggregation. For each reported metric, we first average equally over series with finite metric values within each class, then equally across classes. Metrics are computed on complete series under Series Holdout and on the protocol-specific windows under Segment Holdout. Both protocols use the same 33 dataset–domain–channel classes in the main Random (by class) comparison. A class with no finite score for a metric is omitted from that metric. Paired method comparisons use classes scored by both methods.

Development objective. All experiments use the same scalar quality objective for detector development and candidate selection:

$$Q = 0.8\,\text{VUS-PR} + 0.1\,F_1 + 0.1\,\text{PA-F}_1, \tag{3}$$

where F_1 is point-wise F1 and PA-F_1 is point-adjusted F1. The comparison tables report the individual evaluation metrics. Appendix C.3 describes how Q enters the RL reward.

Threshold calibration. For each submitted candidate, the evaluator calibrates an operating threshold on validation by maximizing Q. Because VUS-PR is independent of any single operating threshold, this calibration affects only the two F1 terms. The selected threshold is frozen and reused for that candidate on internal-test and holdout data, without recalibration using their labels.

Candidate selection. Within a development task, quality is computed separately for each evaluated window and averaged equally within each partition. Let $Q_{\text{val}}(c)$ and $Q_{\text{internal}}(c)$ denote these averages for candidate c. The selection score is

$$S(c) = \begin{cases} \frac{1}{2}\big[Q_{\text{val}}(c) + Q_{\text{internal}}(c)\big], & \text{internal-test aggregates visible,} \\ Q_{\text{val}}(c), & \text{internal-test aggregates hidden.} \end{cases} \tag{4}$$

Internal-test aggregates are visible under Series Holdout and hidden under Segment Holdout and in the hidden-test framework ablations (Appendix B.6); policy training applies the same rule to each task's protocol (Appendix C.3). The framework retains the eligible candidate with the highest selection score. A candidate with missing required scores cannot replace the retained detector. All references to the selection rule in this paper denote $S(c)$.

B AgentAD Framework and Additional Experiments

This appendix describes detector evaluation, diagnostic feedback, and iterative development in AgentAD, followed by the experimental settings, complete results, and analysis procedures. Data partitions and scoring conventions follow Appendix A.

B.1 Detector Programs and Evaluation

A candidate detector is a Python program that assigns an anomaly score to each time point in an input sequence. Each development series is partitioned separately, and the resulting windows remain independent inputs throughout evaluation. The agent may use available labels to guide code revisions, but the detector receives no labels when scoring and runs without language-model calls. Algorithm 1 summarizes the evaluation process.

Algorithm 1 Evaluation of detector programs. $\mathrm{Run}(p, X)$ returns one anomaly score per time point of input $X \in \mathbb{R}^{T \times F}$, with larger scores more anomalous. $\mathcal{W}_s$ contains the windows of development split s, and $\mathcal{H}$ the task's complete holdout series.

Candidate program $\mathrm{Run}(p, X) \to \mathbf{s} \in \mathbb{R}^T$

	Development (each commit)	**Series Holdout** (selected p^*)
Inputs	Each $(X_j, \mathbf{y}_j) \in \mathcal{W}_s$, $s \in \{\mathrm{val}, \mathrm{te}\}$	Each $(X_h, \mathbf{y}_h) \in \mathcal{H}$
Scores	$\mathbf{s}_j \leftarrow \mathrm{Run}(p, X_j)$	$\mathbf{s}_h \leftarrow \mathrm{Run}(p^*, X_h)$
Metrics	$m_j \leftarrow M(\mathbf{s}_j, \mathbf{y}_j)$	$m_h \leftarrow M(\mathbf{s}_h, \mathbf{y}_h)$
Aggregate	$\overline{M}_s = \frac{1}{\lvert\mathcal{W}_s\rvert} \sum_j m_j$	$\overline{M}_{\mathcal{H}} = \frac{1}{\lvert\mathcal{H}\rvert} \sum_h m_h$
Use	Visible-split metrics and validation cases guide development	Final evaluation only

Development-time evaluation. At each commit, the evaluator runs the candidate once on each validation and internal-test window. Metrics are computed separately for each window and averaged equally within each split. Execution takes place in an isolated sandbox without network access. A submission fails if execution raises an error or the output does not contain one finite score per time point. Failed submissions cannot replace the retained detector.

Final evaluation. The selected detector runs separately on each complete holdout series under Series Holdout. Under Segment Holdout, it is evaluated on the internal-test windows. Metrics are aggregated across series and classes as defined in Appendix A.4.

B.2 Diagnostic Feedback and Information Access

The following describes feedback in Full AgentAD. Changes to information access in the framework ablations are specified in Appendix B.6.

Aggregate feedback. For each successfully evaluated candidate, the agent receives the quality score Q and aggregate detection metrics for the visible partitions. These measurements let the agent compare the candidate's performance with earlier submissions. Validation feedback is available in both protocols; internal-test aggregates are returned only under Series Holdout. The quality objective and selection rules are defined in Appendix A.4.

Validation diagnostic cases. The evaluator also returns selected validation slices containing the time-series values, aligned anomaly labels, and the candidate's scores and predictions. The agent receives the numerical data and can analyze them with code; image-capable models may also inspect plots. Cases cover missed events, partial detections, false alarms, and successful detections, with surrounding context to help interpret the signal. They are drawn across error types and series, subject to limits on the number and length of disclosed windows. Recorded predictions allow the agent to compare candidates on the same slice.

Validation disclosure. We deliberately provide validation signals and labels so that the agent can diagnose errors that aggregate metrics alone cannot explain. Disclosed cases remain available for subsequent local experiments. Disclosure is bounded per evaluation, but there is no cumulative limit: successive evaluations may eventually reveal the entire validation set. The prompt asks for reusable detection logic and discourages fitting directly to validation examples, but compliance is not enforced. Cases are chosen for diagnosis, so their frequency does not estimate the full validation error distribution.

Internal-test access. Internal-test windows, labels, pointwise scores, and diagnostic cases remain inaccessible to the agent throughout development. Internal-test performance is therefore measured on data the agent cannot inspect directly. Under Series Holdout, aggregate internal-test metrics are visible and contribute to candidate selection, while complete outer holdout series are reserved for independent final evaluation. Under Segment Holdout, internal-test aggregates are also hidden, and the internal-test windows provide the final evaluation. Appendix C.3 specifies how these partitions are used during policy training.

B.3 Iterative Development and Context Management

Development cycle. The agent explores the signals and available labels in train windows, forms a detection hypothesis, and implements a candidate program. Before a commit, it may test an idea locally or check that the program runs. At each commit, the agent submits its current detector program for evaluation, then uses the measurements and disclosed validation cases to decide what to revise. The framework retains the eligible candidate with the highest selection score $S(c)$ throughout the trajectory (Equation 4).

Experiment history and context. Code, measurements, and disclosed diagnostic cases remain available across commits. Reasoning is retained within the current commit segment and removed from subsequent model context when that segment closes. When the conversation approaches the context limit, the framework preserves the initial task and recent complete turns and summarizes older experiments where possible. Only the configured number of recent images remain in the model's context; the archived trajectory retains the originals. These limits govern the current context, while validation disclosure is governed by the rules in Appendix B.2.

Budgets and termination. Each trajectory has limits on submissions, model turns, and tool calls. Reminders encourage a submission after the recommended number of turns and announce when only one commit remains; they do not impose a hard deadline between commits. Development ends with a final submission or when the development budget is exhausted, and the retained candidate is delivered. Stalled quality does not trigger early stopping in the main comparison. Table 9 gives the shared budget and context settings.

B.4 Developer Models and Experimental Settings

Single-path development. Each AgentAD row in the main detector comparison uses one development trajectory per class. Table 8 lists the models, input modalities, and reasoning settings used in the main comparison, framework ablations, and policy training. GLM-5.3 receives text only; the other listed models may also receive diagnostic plots. Table 9 gives the shared development configuration. Data access and candidate selection follow the protocols in Appendix A.3, and training-specific settings are given in Appendix C.

Run statistics. Table 10 reports token use, commits, wall-clock time, and estimated API costs for the runs behind Table 1. Actual token use and runtime vary across models under the same budget limits.

B.5 Complete Results and Coverage

Methods differ in their coverage of the series summarized in Table 1. We report this coverage alongside the complete developer comparison and per-class results, using the aggregation rules in Appendix A.4.

Table 8: **Developer models and reasoning settings.** Rows are grouped by experiment: main comparison (General), framework ablations, and policy training (RL).

Model	Architecture	Size (B)	Effort	Input
General				
Qwen3.5-4B	Dense	4	xhigh	Text + image
Qwen3.5-9B	Dense	9	xhigh	Text + image
Qwen3.5-35B-A3B	MoE	35	xhigh	Text + image
Qwen3.8-27B	Dense	27	xhigh	Text + image
DeepSeek-V4.1-Flash	MoE	552^a	max	Text + image
GLM-5.3-Flash	MoE	320	max	Text + image
GLM-5.3	MoE	753^b	max	Text
Ablation				
Qwen3.8-27B	Dense	27	medium	Text + image
RL				
GLM-5.3-Flash (teacher)	MoE	320	medium	Text + image
Qwen3.5-35B-A3B (policy)	MoE	35	xhigh	Text + image

Sizes are total parameters in billions. Qwen3.5-35B-A3B activates 3B parameters and GLM-5.3-Flash activates 18B. [a]DeepSeek-V4.1-Flash reports a 552B backbone plus 196B Engram memory, with 8B/16B active parameters during prefill/decode. [b]Listed size of the released GLM-5.3 checkpoint. Effort labels are model-specific and do not imply matched compute. The RL policy row covers SFT, DAPO, and C-DAPO.

Table 9: **AgentAD development configuration.** Settings shared by all developers in the main comparison.

Setting	Configuration
Search and budgets	
Development search	One trajectory per class
Commit attempts	At most 10 per trajectory
Model turns / tool calls	At most 400 / 400 per trajectory
Commit-pace reminders	After 30 turns before the first commit; 20 turns between later commits
Context and observations	
Reasoning retention	Reasoning within the current commit segment
Image history	At most 200 recent images in model-visible context
Developer and effort	Experiment-specific settings in Table 8
Objective and delivery	
Quality objective	$Q = 0.8\,\mathrm{VUS\text{-}PR} + 0.1\,F_1 + 0.1\,\mathrm{PA\text{-}F_1}$
Candidate retention	Eligible candidate with the highest $S(c)$ (Equation 4)
Early stopping	Quality-stall stopping disabled
Failed commits	Cannot replace the retained candidate
Unscored runs	No valid delivered program

F_1 is point-wise F1 and $\mathrm{PA\text{-}F_1}$ is point-adjusted F1. Commit-pace reminders are advisory and impose no hard deadline.

Table 10: **AgentAD run statistics.** Mean cost per development run for the developers in Table 1.

Model	Tokens (M)		Commits		Time (min)		Equiv. API ($)	
	Ser.	Seg.	Ser.	Seg.	Ser.	Seg.	Ser.	Seg.
Qwen3.5-4B	2.09	4.52	2.64	2.63	18.5	17.9	–	–
Qwen3.5-9B	0.83	2.21	2.76	3.13	12.1	22.9	0.014	0.033
Qwen3.5-35B-A3B	2.06	2.83	6.48	6.31	19.0	25.6	0.109	0.154
Qwen3.8-27B	4.33	4.53	4.88	5.50	87.4	69.0	0.587	0.642
DeepSeek-V4.1-Flash	5.48	5.02	5.33	5.25	69.6	97.2	0.112	0.110
GLM-5.3-Flash	4.53	4.50	7.06	6.69	37.8	105.8	0.195	0.201
GLM-5.3	4.58	4.92	4.50	4.69	137.9	187.9	1.673	1.837

Ser. and Seg. denote Series and Segment Holdout. Values are means over all 33 classes. Time is wall-clock and includes tool execution and candidate evaluation. API costs are estimated from publicly available prices and provided for reference only; dashes indicate unavailable estimates.

Established methods. Both protocols use the same 280 series in 33 classes, with complete-series or suffix scoring as defined in Appendix A.3. A series contributes to a method's result only when the method can be applied and the corresponding metric is finite.

A method covers a class when it can be applied to at least one series in that class. Under Series Holdout, TSPulse-FT, CARLA, and AERCA cover 31, 31, and 25 of the 33 classes. Under Segment Holdout, TSPulse-FT and AERCA cover 28 and 25 classes, and Left covers all 33 classes from 159 series. Every other established method in the main table covers all 33 classes under both protocols.

Table 11 reports the number of successfully evaluated series and classes with complete evaluation coverage under each protocol.

Table 11: **Evaluation coverage of established neural detectors.**

Method	Series Holdout		Segment Holdout	
	Valid series	Full classes	Valid series	Full classes
Non-Transformer neural				
PaAno	280/280	33/33	280/280	33/33
TSPulse-FT	270/280	31/33	179/280	23/33
TSPulse-ZS	280/280	33/33	280/280	33/33
KAN-AD	280/280	33/33	280/280	33/33
xLSTMAD	280/280	33/33	280/280	33/33
DADA	280/280	33/33	280/280	33/33
ScatterAD	280/280	33/33	280/280	33/33
AERCA	139/280	25/33	135/280	24/33
CARLA	270/280	31/33	280/280	33/33
Transformer				
Time-RCD	280/280	33/33	280/280	33/33
Left	280/280	33/33	159/280	26/33
AxonAD	280/280	33/33	280/280	33/33
CrossAD	280/280	33/33	280/280	33/33

Valid series are holdout series to which a method can be applied; a class is full when all of its series are valid. The seven classical methods are fully covered under both protocols.

AgentAD. A class enters the mean when its development run finishes and the metric is finite. All seven developer models in Table 1 have a class-mean VUS-PR for each of the 33 classes under both protocols.

Complete developer comparison. Table 12 compares the seven developer models under Random (by class) and the supplementary Eva setting. Random (by class) uses the class-wise split in Appendix A.2. Eva keeps the fixed TSB-AD series membership and groups tasks by dataset: under Series Holdout its series serve as complete holdout series (16 eligible tasks), and under Segment Holdout they serve as development series (40 tasks). Scores are averaged over series within each task

and then equally across tasks. In both settings and under both protocols, Qwen3.8-27B, DeepSeek-V4.1-Flash, GLM-5.3-Flash, and GLM-5.3 exceed all three Qwen3.5 developers in VUS-PR.

Table 12: **AgentAD results across data partitions.**

Model	VUS-PR		VUS-ROC		AUC-ROC		Range-F1	
	Ser.	Seg.	Ser.	Seg.	Ser.	Seg.	Ser.	Seg.
Random (by class)								
Qwen3.5-4B	0.4659	0.5017	0.8184	0.8046	0.7829	0.7567	0.1654	0.1849
Qwen3.5-9B	0.4688	0.4872	0.8123	0.7919	0.7785	0.7406	0.1623	0.1863
Qwen3.5-35B-A3B	0.4851	0.5008	0.8161	0.8052	0.7849	0.7511	0.1820	0.1926
Qwen3.8-27B	0.5312	0.5891	0.8586	0.8443	0.8283	0.8021	0.2259	0.2024
DeepSeek-V4.1-Flash	0.6056	0.5903	**0.8958**	0.8469	**0.8786**	0.8024	0.2235	0.1970
GLM-5.3-Flash	0.5668	0.5716	0.8595	0.8337	0.8410	0.7927	0.2262	0.2201
GLM-5.3	**0.6297**	**0.6570**	0.8817	**0.8803**	0.8709	**0.8620**	**0.2493**	**0.2352**
Eva								
Qwen3.5-4B	0.4896	0.5954	0.8294	0.8322	0.8049	0.7790	0.1687	0.2324
Qwen3.5-9B	0.4919	0.6231	0.8376	0.8403	0.8135	0.7874	0.1615	0.2421
Qwen3.5-35B-A3B	0.4769	0.6252	0.8108	0.8386	0.7905	0.7847	0.1682	0.2276
Qwen3.8-27B	0.5731	0.6998	0.8674	0.8846	0.8563	0.8452	0.1963	0.2305
DeepSeek-V4.1-Flash	0.5794	0.7124	0.8857	0.8834	0.8753	0.8464	0.2110	0.1955
GLM-5.3-Flash	**0.6276**	0.7112	**0.9311**	**0.8887**	**0.9241**	**0.8524**	0.2247	0.2405
GLM-5.3	0.6150	**0.7266**	0.9210	0.8828	0.9185	0.8464	**0.2296**	**0.2481**

Higher is better; bold marks the best value in each column within each partition. Random (by class) is the 33-class split of Table 1; Eva groups the fixed TSB-AD membership by dataset (16 Series Holdout and 40 Segment Holdout tasks).

Per-class results. Figures 6 and 7 show VUS-PR for individual methods and classes under the two protocols. Each cell is a class mean, and hatched cells indicate unavailable evaluations. The best AgentAD developer exceeds the best established method in 26 of the 33 classes under Series Holdout and in 21 under Segment Holdout; GLM-5.3 alone does so in 23 and 18 classes.

B.6 Framework Ablation Settings

All variants use Qwen3.8-27B with medium reasoning effort on the seven Series Holdout classes in Appendix A.2. They share the outer data split, evaluator, quality objective Q, generation limits, and access to train labels. Runs with iterative feedback allow up to 10 commits. Table 13 summarizes the data available to each variant, its feedback, and its selection and submission rules. Paired comparisons use classes scored by both variants.

Iterative development. *Full AgentAD* follows the feedback and context rules above and retains the eligible detector with the highest $S(c)$. *First Commit* reports the retained detector after the first submission attempt of that same run. A failed first attempt is not skipped, and the model is not separately instructed to solve the task in one attempt. *Python-only Development* permits local code experiments on complete development series and available labels but allows only one final submission, without intermediate evaluation feedback. Because this variant exposes internal-test regions, it has no independent Segment Holdout result.

Diagnostic evidence and data access. *No Diagnostic Cases* preserves train exploration and aggregate validation and internal-test feedback, including error counts, but removes validation slices, their labels, pointwise predictions, and case comparisons. This comparison tests whether labeled validation slices improve development beyond aggregate feedback alone. *No Test Feedback* retains the validation cases but hides internal-test aggregates and selects candidates using validation quality alone. *Full Validation Access* additionally exposes the complete validation signals and labels; internal-test aggregates remain hidden, and selection still uses validation alone. Comparing these last two variants isolates the effect of expanding validation access from selected slices to complete windows. Comparing Full AgentAD with No Test Feedback changes both aggregate feedback and candidate selection.

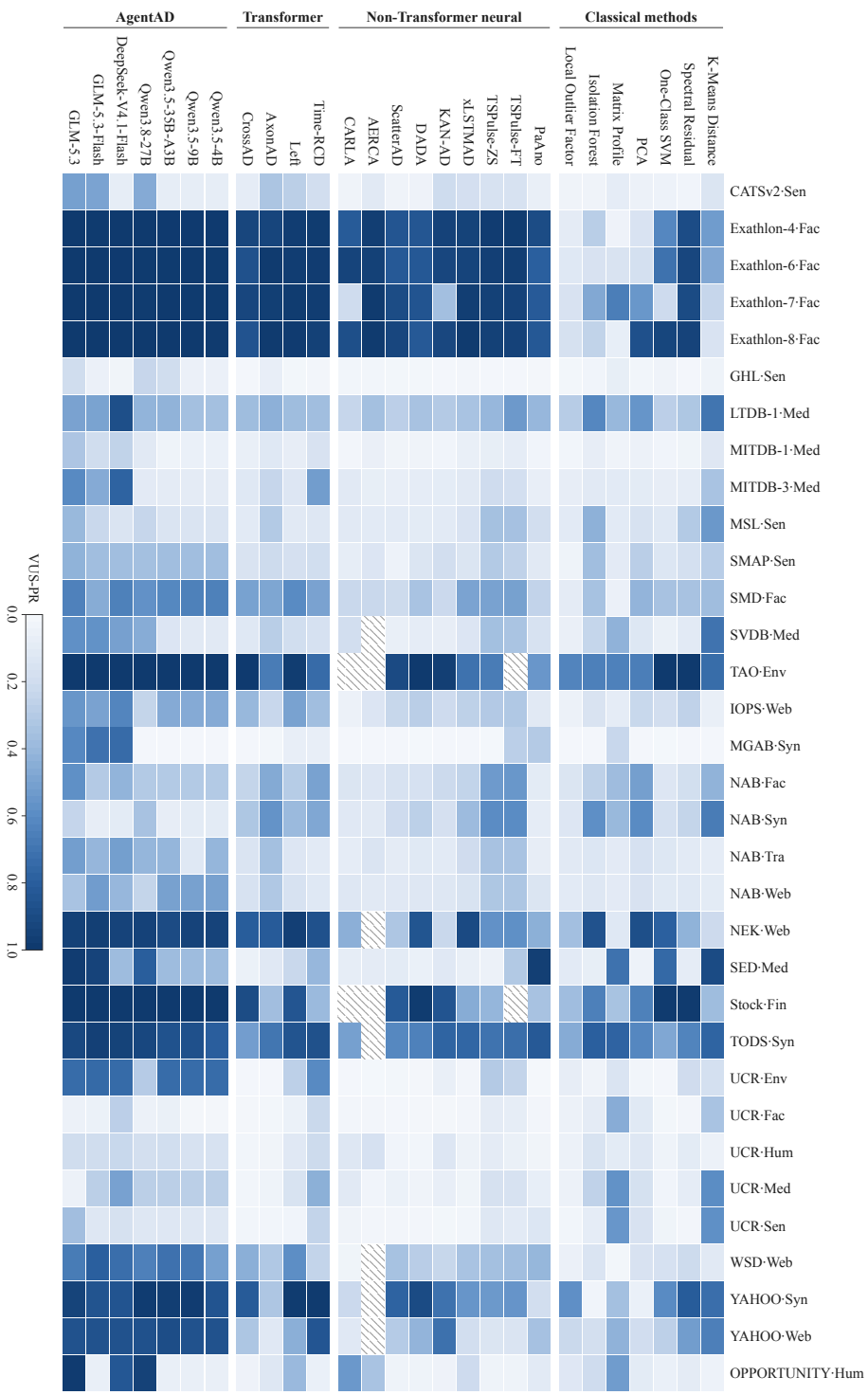

Figure 6: **Per-class VUS-PR under Series Holdout.** Each cell is a class-mean VUS-PR on one of the 33 evaluated classes; hatched cells were not evaluated.

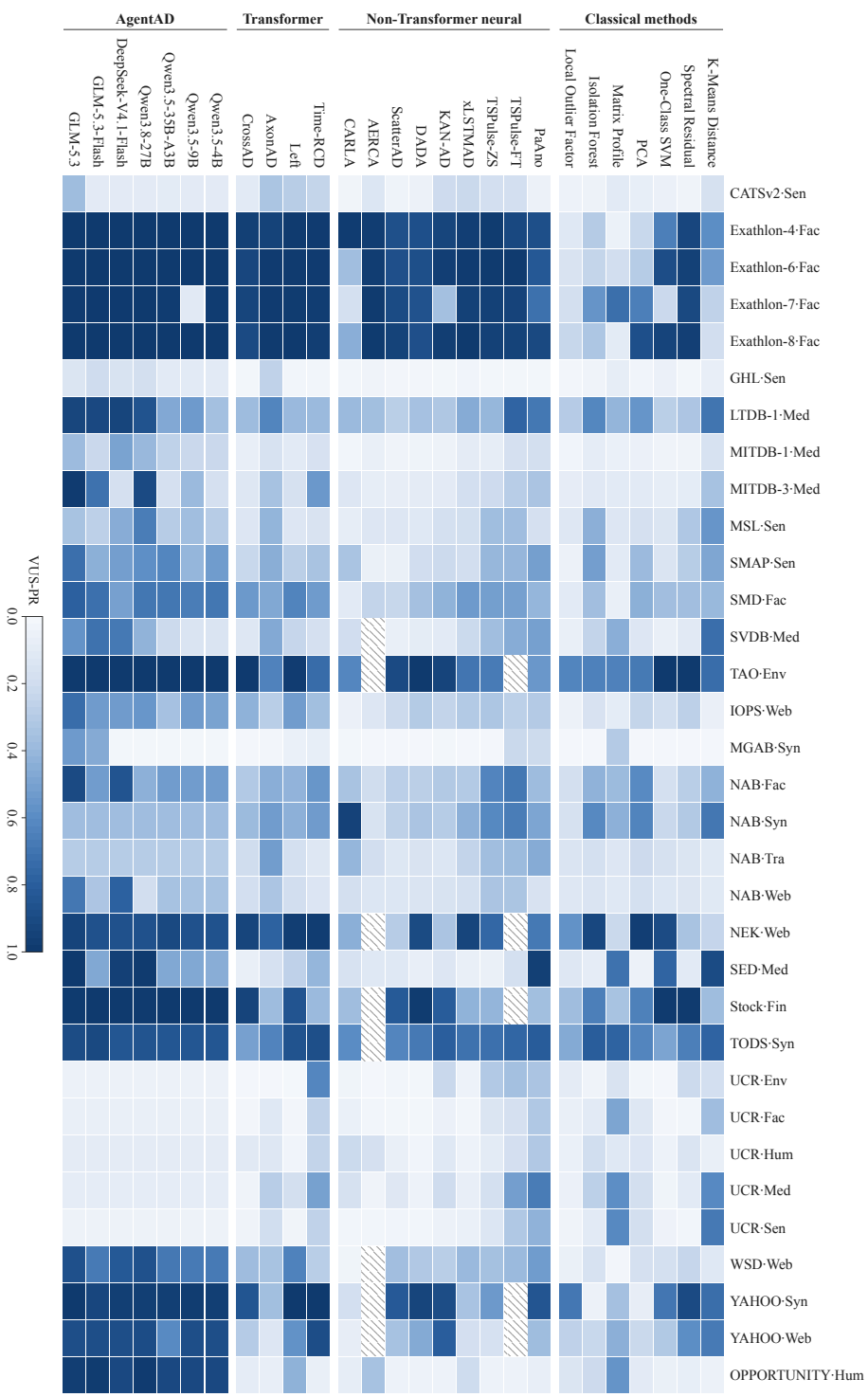

Figure 7: **Per-class VUS-PR under Segment Holdout.** Established methods are scored on archive suffixes; AgentAD on internal-test windows.

Table 13: **Framework ablation settings.** Evidence available to the agent, aggregate feedback, and selection in each variant.

Setting	Signals and labels	Aggregate feedback	Selection and submissions
Full AgentAD	Train; selected validation slices	Validation and internal test	Mean validation and internal-test Q; up to 10 commits
First Commit	As Full AgentAD	As Full AgentAD	Retained candidate after the first attempt of the full run
Python-only Development	Complete development series and available labels	No intermediate evaluation feedback	One final submission
No Diagnostic Cases	Train only	Validation and internal test	Mean validation and internal-test Q; up to 10 commits
No Test Feedback	Train; selected validation slices	Validation only	Validation Q; up to 10 commits
Full Validation Access	Complete train and validation windows	Validation only	Validation Q; up to 10 commits

Selected validation slices contain signals, labels, and candidate predictions (Appendix B.2). Internal-test windows remain hidden except in Python-only Development, and no variant accesses outer holdout series during development.

Table 14 reports all four metrics; Table 3 presents VUS-PR and Range-F1 in the main text. Full AgentAD is best on VUS-PR and Range-F1, whereas No Diagnostic Cases is best on VUS-ROC and AUC-ROC despite a VUS-PR 0.1355 lower. Aggregate feedback alone thus preserves overall score ranking but gives less precise detection of labeled anomalies.

Table 14: **Framework ablations on all four metrics.**

Setting	VUS-PR	VUS-ROC	AUC-ROC	Range-F1
Full AgentAD	**0.5864**	0.8010	0.7813	**0.3416**
First Commit	0.3942	0.7495	0.7238	0.2593
Python-only Development	0.3870	0.7736	0.7314	0.3145
No Diagnostic Cases	0.4509	**0.8183**	**0.7885**	0.3138
No Test Feedback	0.5528	0.8144	0.7811	0.3118
Full Validation Access	0.5137	0.8065	0.7763	0.2576

Holdout class means on the seven ablation classes. Higher is better; the best value in each column is bold. Settings are defined in Table 13.

B.7 Development Dynamics and Feedback Analysis

Development curves. Figure 5 follows the detector retained after each submission attempt k in a completed trajectory. Selected@k reports its holdout VUS-PR. Development Best@k reports the same detector's mean validation and internal-test VUS-PR when internal-test aggregates are visible, and its validation VUS-PR otherwise. This is the performance available to the agent for the candidate selected by $S(c)$ (Equation 4), not a running maximum of VUS-PR. Oracle Best@k is the maximum holdout VUS-PR among eligible candidates observed through attempt k within that task.

We compute each curve within tasks before averaging across them. The development–holdout gap compares Development Best@k with Selected@k for the same detector. Oracle Best@k uses the same tasks as Selected@k, and the shaded gap is their difference: the holdout performance gap between the best candidate observed so far and the retained detector.

The attempt index includes failed and rewound submissions. A failed attempt leaves the previous detector in place. When a trajectory ends early, its final detector is carried forward for comparison at a common attempt budget; a trajectory with no valid detector remains a failure. These curves use

prefixes of completed runs, rather than separate runs assigned different budgets. Oracle values are computed retrospectively and never guide development or selection.

Development–holdout correlations. Each scored commit contributes a pair: the candidate's development and outer-holdout values of the same metric. The development value is the mean of validation and internal-test values when internal-test aggregates are visible, and the validation value otherwise. We compute Pearson and Spearman correlations separately within each task, then average the task-level coefficients equally, without pooling candidates across tasks.

A task is omitted if it has fewer than three scored pairs or either score is constant after rounding to six decimals. Scores are not normalized before computing correlations. Table 15 reports all four metrics and the number of eligible tasks; Table 4 presents VUS-PR and Range-F1 in the main text. All outer-holdout scores used in these analyses are calculated after the trajectories and selection decisions are frozen.

Table 15: **Development–Holdout correlations for all four metrics.**

Setting	VUS-PR		VUS-ROC		AUC-ROC		Range-F1	
	r	ρ	r	ρ	r	ρ	r	ρ
Full AgentAD	0.668	0.507	0.085	0.080	0.937	0.873	0.758	0.789
No Test Feedback	0.062	0.202	−0.181	0.063	0.249	0.128	0.769	0.823
Full Validation Access	0.238	0.247	0.246	0.247	0.261	0.247	0.431	0.433
No Diagnostic Cases	0.834	0.602	0.649	0.465	0.796	0.589	0.333	0.221

Entries average within-task Pearson (r) and Spearman (ρ) correlations over eligible tasks (Appendix B.7). Full Validation Access has five eligible tasks for every metric; the other settings have seven for VUS-PR and VUS-ROC, five for AUC-ROC, and seven (Full AgentAD) or six for Range-F1.

C Developer Policy Training

This appendix describes the training tasks, SFT initialization, reinforcement learning objective, and evaluation of developer policies. Data partitions and information access follow Appendix A, and the development process follows Appendix B.

C.1 Training Tasks and Evaluation Separation

SFT and RL use four task pools: Series Holdout and Segment Holdout under Random (by class) and Eva. Random (by class) follows the main comparison's seed-42 split (Appendix A.2). Eva uses the fixed TSB-AD membership and dataset-level task groups in Table 12. Each task asks the agent to develop a detector for a class or dataset, which may contain several series.

Policy training and evaluation are separated by class. All series from the seven classes in Appendix A.2 are excluded from every SFT and RL task pool, including the dataset-level Eva tasks. Both Series Holdout and Segment Holdout evaluate developer policies on these same seven classes, so this exclusion applies to both protocols. The framework ablations also use these classes. After exclusion, each Random pool contains 26 tasks, and the Eva pools contain 16 Series Holdout tasks and 40 Segment Holdout tasks, giving 108 tasks shared by SFT and RL.

Within a training task, data use follows its holdout protocol. Series Holdout keeps complete holdout series out of detector development. Under Segment Holdout, the agent develops detectors using train windows and validation feedback, while internal-test feedback is hidden. Internal-test windows used to compute RL rewards are training data, regardless of whether their metrics are visible to the agent. Appendix C.5 specifies the evaluation tasks and settings.

C.2 Teacher Trajectories and SFT Initialization

Teacher trajectories. We collect detector-development trajectories with GLM-5.3-Flash at reasoning effort `medium`. The teacher uses the shared quality objective Q and selection score $S(c)$

from Appendix A.4. Its feedback includes internal-test aggregates under Series Holdout and validation feedback alone under Segment Holdout. Each task contributes four trajectories, with at most 10 commits per trajectory and a 262,144-token context limit. Context management follows Appendix B.3; these collection limits are specified separately from the main comparison's development budget.

Supervised targets. Each trajectory is converted into assistant-turn training examples. Generated reasoning, code, and tool calls are supervision targets, while task instructions and observations provide context and are masked from the loss. Examples preserve the context available to the teacher at each turn, including the reasoning-retention rule at commit boundaries.

SFT initialization. Qwen3.5-35B-A3B-SFT starts from the Qwen3.5-35B-A3B base model and is trained with completion-only cross-entropy. Table 16 gives the LoRA and optimization settings. Its final training checkpoint initializes both DAPO and C-DAPO. Figure 8 reports the distributions of commit counts, tool calls, and peak context length across teacher development trajectories.

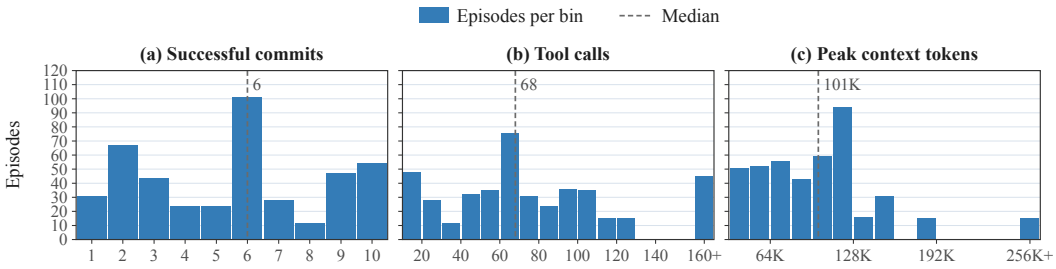

Figure 8: **Teacher-trajectory distributions.** Dashed lines mark medians. The rightmost bars also include episodes with 160 or more tool calls or more than 256K context tokens; K denotes 1,024 tokens.

C.3 Reinforcement Learning Objective and Credit Assignment

DAPO and C-DAPO start from the same SFT checkpoint and continue training their LoRA adapters in the AgentAD development environment. They use the same training tasks, feedback rules, development budgets, reward function, and optimization settings. C-DAPO adds a commit-level auxiliary advantage; DAPO sets its weight to $\alpha = 0$. Appendix C.4 gives the sampling and optimization procedure.

Selection and reward partitions. Candidates are retained according to $S(c)$ (Equation 4), which averages validation and internal-test quality when internal-test aggregates are visible and uses validation quality otherwise. C-DAPO's subsequent gains track this selection score. The detector-quality component of the terminal reward is the retained candidate's internal-test Q. Complete holdout series within Series Holdout tasks do not contribute to this reward.

Trajectories and commit segments. For a task q, we independently sample N trajectories from a policy π_{old} that remains fixed throughout collection. Trajectories share the initial task context and development limits. A commit segment contains the generated reasoning, code, and tool calls leading to a submission attempt and its feedback. It may span several model turns, and failed submissions also close segments. Generated content after the last submission forms a terminal suffix.

Each generated token is trained under the context in which it was produced, following Appendix B.3. If a model turn contains multiple commits, its generated tokens belong to the first commit; later attempts keep their original indices without duplicating token supervision. Prompts, observations, and images provide context but are not training targets.

Terminal reward. Let c_i^* be the candidate retained at the end of trajectory i, and let $Q_{\text{internal}}(c_i^*)$ be its mean internal-test quality. With T_i submission attempts and total generated-length penalty P_i,

Table 16: **SFT training configuration.** Qwen3.5-35B-A3B-SFT is trained on the teacher corpus in Appendix C.2.

Setting	Configuration
Data	
Teacher / reasoning effort	GLM-5.3-Flash / medium
Training tasks	108 retained tasks (Appendix C.1)
Trajectories per task	4
Supervision targets	Reasoning, code, and tool calls; prompts and observations masked
Context / reasoning retention	Recorded model-visible context / current commit segment
Input modalities	Text and images
Maximum sequence length	262,144 tokens; longer examples truncated
Initialization	
Initial model	Qwen3.5-35B-A3B
LoRA rank / α_{LoRA} / dropout	16 / 32 / 0
Trainable modules	Language attention and MLP; vision encoder and merger frozen
Optimization	
Loss	Completion-only cross-entropy
Batch	64 assistant-turn examples per step; length-grouped batches
Epochs	3
Optimizer / learning rate	AdamW / 5×10^{-5}
Adam β_1 / β_2 / ϵ	0.9 / 0.999 / 10^{-8}
Schedule	Cosine, 5% linear warmup
Weight decay / gradient-norm limit	0 / 1
Random seed	42
Final checkpoint	Last training checkpoint; no selection on held-out evaluation classes

Series Holdout trajectories include visible internal-test aggregates; Segment Holdout trajectories do not.

the terminal reward is

$$R_i = r_i^{\mathrm{out}} - \lambda T_i - P_i, \qquad r_i^{\mathrm{out}} = Q_{\mathrm{internal}}(c_i^*). \tag{5}$$

If no valid candidate is retained, $r_i^{\mathrm{out}} = -1$. Failed attempts count toward the commit cost but cannot replace a valid candidate. An unfinished terminal suffix adds no commit charge.

For the generated segments $\mathcal{Z}_i$, including any terminal suffix, $P_i = \sum_{z \in \mathcal{Z}_i} P_{\mathrm{len}}(|z|)$, where

$$P_{\mathrm{len}}(L) = \eta \ \mathrm{clip}\left(\frac{L - (L_{\max} - L_c)}{L_c}, 0, 1\right). \tag{6}$$

Here $|z|$ counts generated tokens. The penalty rises from zero to η over $[L_{\max} - L_c, L_{\max}]$ and then saturates. This threshold controls length penalties and masking during training; it does not end the development run.

Trajectory advantage. Let μ_R and σ_R be the population mean and standard deviation of terminal rewards in one task group. Groups with $\sigma_R < \delta_R$ are excluded. For retained groups,

$$A_i^{\mathrm{traj}} = \frac{R_i - \mu_R}{\max(\sigma_R, \sigma_{\min})}. \tag{7}$$

The scale floor $\sigma_{\min}$ limits amplification of small reward differences. With $\sigma_{\min} = \delta_R$, retained groups use their own standard deviation.

Subsequent gains in selection quality. Let $v_{i,t}$ be the running-best $S(c)$ through commit boundary t, and v_i^{final} its value at termination. The initial value $v_{i,0}$ is the score of any initially retained candidate, or zero if none exists; training supplies no additional initial detector. A lower-scoring or invalid submission leaves the retained score unchanged. The gain for segment t is

$$g_{i,t} = v_i^{\mathrm{final}} - v_{i,t-1},$$

as defined in Equation 1. It includes improvements after the current segment, allowing an unsuccessful experiment to receive credit when later revisions improve the detector. Returning to a previously attained score adds no new gain. This signal measures observed subsequent progress, not a segment's causal contribution.

Commit-level normalization. At commit index t, let $\mathcal{V}_t$ contain trajectories with an eligible, unmasked segment. Such a segment contains generated tokens and is neither truncated nor overlong. Trajectories that end before t do not participate, and masking does not renumber commit indices. Using the population moments $\mu_{g,t}$ and $\sigma_{g,t}$ of gains over $\mathcal{V}_t$, we compute

$$A^{\mathrm{com}}_{i,t} = \mathrm{Norm}_t(g_{i,t}) = \begin{cases} (g_{i,t} - \mu_{g,t}) / \max(\sigma_{g,t}, \sigma_{\min}), & i \in \mathcal{V}_t,\ |\mathcal{V}_t| \geq 2,\ \sigma_{g,t} \geq \delta_c, \\ 0, & \text{otherwise.} \end{cases} \tag{8}$$

The normalized advantage compares progress across trajectories at the same commit position. A zero raw gain can receive a negative advantage when other trajectories improve. Too few eligible trajectories or low gain variance disables only the auxiliary term, without excluding an otherwise valid task group.

Advantage combination and direction protection. Equation 2 adds the auxiliary term to the trajectory advantage:

$$\widetilde{A}_{i,t} = A^{\mathrm{traj}}_i + \alpha A^{\mathrm{com}}_{i,t}.$$

We then apply

$$A_{i,t} = \begin{cases} 0, & A^{\mathrm{traj}}_i \widetilde{A}_{i,t} < 0, \\ \widetilde{A}_{i,t}, & \text{otherwise.} \end{cases} \tag{9}$$

The auxiliary term can strengthen or attenuate an update but cannot reverse a nonzero trajectory-advantage direction. If $A^{\mathrm{traj}}_i = 0$, it may determine the update direction. Every eligible generated token in a commit segment shares $A_{i,t}$; terminal suffixes receive only A^{traj}_i. Ineligible segments remain masked from optimization.

Credit assignment. Figure 3 applies the calculation to four trajectories starting at $v_{i,0} = 0.60$, with $Q_{\mathrm{internal}}(c^*_i) = v^{\mathrm{final}}_i - 0.01$, zero length penalties, and $\alpha = 0.3$. Training uses $\alpha = 0.1$. Later improvements assign credit to an earlier unsuccessful experiment, and commits after progress stalls receive lower relative credit.

C.4 Sampling, Optimization, and Training Configuration

Task-group sampling. We retain complete task groups that pass the reward-variance threshold in Appendix C.3 and collect additional groups under the same frozen policy to fill the batch. Collection ends when the target batch is full or after M rounds, including the initial round; a partial batch is accepted. Infrastructure failures, missing required measurements, or inconsistent initial selection scores exclude an entire group. Invalid submissions produced by the policy remain scored outcomes.

Token-level objective. Let z index generated segments in the accepted batch, including terminal suffixes, and let A_z be their assigned advantages. For generated token $y_{z,k}$ with context $h_{z,k}$, define

$$\rho_{z,k} = \frac{\pi_\theta(y_{z,k} \mid h_{z,k})}{\pi_{\mathrm{old}}(y_{z,k} \mid h_{z,k})}.$$

With $W = \sum_{z,k} w_{z,k} > 0$, the loss is

$$\mathcal{L}_{\text{C-DAPO}} = -\frac{1}{W} \sum_{z,k} w_{z,k} \min\{\rho_{z,k} A_z,\ \mathrm{clip}(\rho_{z,k}, 1 - \varepsilon_{\mathrm{low}}, 1 + \varepsilon_{\mathrm{high}}) A_z\}. \tag{10}$$

Eligible generated tokens have weight one. Truncated segments, segments with $|z| > L_{\max}$, and padding have weight zero; prompts and observations are never targets. Masking a segment leaves earlier eligible segments trainable and does not change environment feedback. The denominator covers the full optimizer batch, giving each eligible token equal weight. We perform one update per collection without a KL term and skip the update when $W = 0$.

Matched training settings. Table 17 gives the configuration shared by DAPO and C-DAPO, including generation budgets and optimizer settings. Setting $\alpha = 0$ yields the trajectory-level DAPO comparison with the same reward, filtering, normalization, and token-level objective. Both policies continue from the SFT checkpoint in Appendix C.2.

Table 17: **RL training configuration.** DAPO and C-DAPO share every setting except the commit-level weight α. Reward and optimization symbols are defined in Appendices C.3 and C.4.

Setting	Configuration
Initialization	
Initial policy	Qwen3.5-35B-A3B-SFT; its LoRA adapter continues training
LoRA rank / α_{LoRA} / dropout	16 / 32 / 0
Trainable modules	Language attention and MLP; vision encoder and merger frozen
Rollout	
Training tasks	108 seed-42 tasks from Appendix C.1
Trajectories per task N	8
Reasoning effort / retention	xhigh / current commit segment
Sampling	Temperature 1, top-p 1; no top-k or repetition penalty
Commits / turns / tool calls	At most 10 / 64 / 400 per trajectory
Context window	131,072 tokens
Generated tokens	At most 65,536 per model call and 131,072 per trajectory
Reward and advantage	
Quality objective	$Q = 0.8\,\mathrm{VUS\text{-}PR} + 0.1\,F_1 + 0.1\,\mathrm{PA\text{-}F_1}$
Candidate selection / reward	$S(c)$ (Equation 4) / internal-test Q
Commit cost λ / length penalty η	2×10^{-4} / 2×10^{-3}
Segment limit $L_{\max}$ / penalty ramp L_c	131,072 / 8,192 tokens
Variance thresholds δ_R / δ_c	2×10^{-3} / 2×10^{-3}
Advantage scale floor $\sigma_{\min}$	2×10^{-3}
Commit-level weight α	0 (DAPO); 0.1 (C-DAPO)
Optimization	
Target batch	16 task groups (128 trajectories) per update
Collection rounds M / updates	3 / 1 per collection
Clipping $\varepsilon_{\mathrm{low}}$ / $\varepsilon_{\mathrm{high}}$	0.2 / 0.28
KL coefficient	0
Optimizer / learning rate	AdamW / 10^{-6}, constant, no warmup
Gradient-norm limit	1
Epochs / evaluated checkpoints	3 / end of each epoch
Random seed	42

The seven evaluation classes are excluded from all training tasks. Groups filtered by δ_R are resampled within the M collection rounds, and truncated or overlong segments are masked from the loss.

C.5 Evaluation of Developer Policies

Evaluation tasks and coverage. Both holdout protocols in Table 2 use the seven classes excluded from policy training (Appendix C.1). Series Holdout scores complete holdout series, while Segment Holdout scores internal-test windows from separate development runs within the same classes, following Appendix A.3. Scores are averaged over series within each class and then equally across classes. Figure 4 and the iteration analysis in Table 5 report Series Holdout performance on this same class set.

Development settings and checkpoints. All developer policies are evaluated with the same AgenTAD settings and execution limits as the main comparison, including the 10-commit budget in Table 9. Model-specific reasoning effort and input modalities follow Table 8. Candidate selection uses the shared objective Q and protocol-specific score $S(c)$.

The main policy comparison in Table 2 and the iteration analysis in Table 5 use the final-epoch checkpoints of DAPO and C-DAPO. Figure 4 additionally evaluates checkpoints after each training epoch to show training progress. Scores on the seven evaluation classes are not used to choose checkpoints or hyperparameters.

Evaluation scores and training signals. Evaluation scores on the seven reserved classes do not enter policy training, its terminal rewards, or either advantage term. Within each evaluation task, detector development and candidate selection still follow the information-access rules of its protocol: Series Holdout reports performance on complete holdout series, and Segment Holdout reports performance on hidden internal-test windows. Training rewards and advantages use only the training tasks described in Appendices C.1 and C.3.

Improvement during development. Table 5 reports holdout VUS-PR for the first scored submission and the final retained detector in each completed trajectory. Here, First is the first submission with a score; First Commit in the framework ablations uses the first attempt, which may fail (Appendix B.6). Improvement is Selected minus First within each task, averaged across the same seven classes. Commits and development tokens are per-run means. The improvement should be interpreted together with the starting score and actual cost, since a shared budget limit does not imply equal computation.

D Series Visualizations and Detector Programs

This appendix presents visualizations of benchmark series and their development partitions, followed by complete detector programs produced in our experiments.

D.1 Series Visualization

The series visualization interface displays time-series values together with labeled anomaly intervals. Figure 9 shows benchmark series in this format, with values normalized for display and anomaly intervals marked along the time axis.

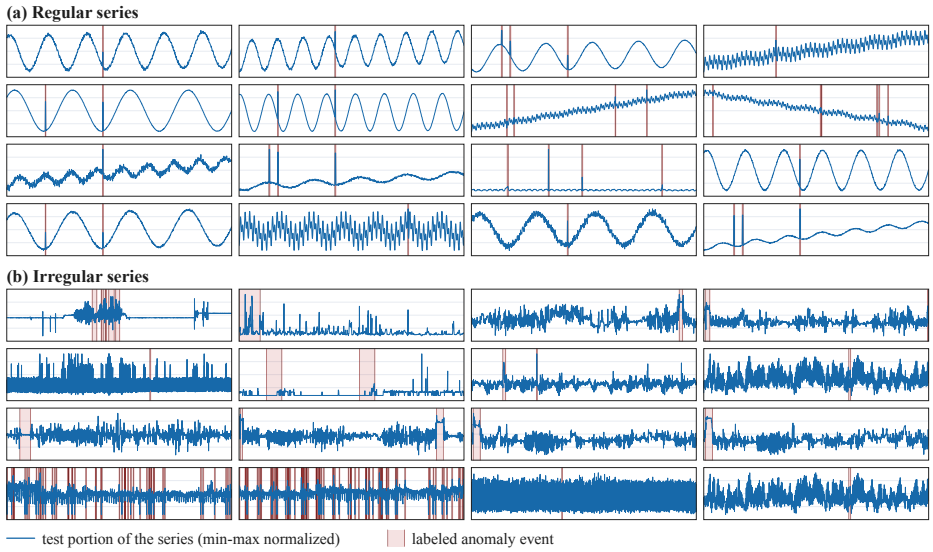

Figure 9: **Visualization of benchmark time series.** Each cell shows the test portion of one series. Values are min-max normalized for display, and shaded bands mark labeled anomaly events.

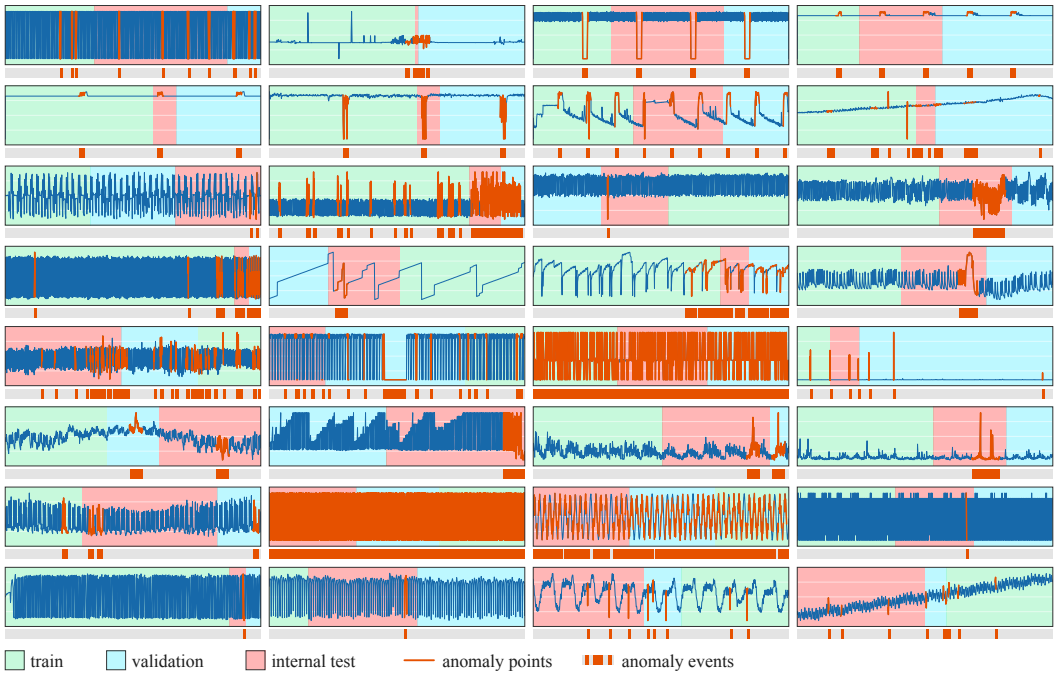

Figure 10: **Development partitions of benchmark series.** One development series from each of 32 classes under Random (by class). Train, validation, and internal-test windows are shown together with the signals and labeled anomaly events.

D.2 Development Partitions

Figure 10 shows the train, validation, and internal-test windows for 32 development series under Random (by class), one from each of 32 classes. The plots display the partition boundaries alongside the signals and anomaly annotations. The partitioning rules are defined in Appendix A.2.

D.3 Complete Detector Programs

Figures 11–14 reproduce the complete source of four selected detector programs: Qwen3.8-27B on SMAP, GLM-5.3-Flash on GECCO, GLM-5.3 on MITDB, and DeepSeek-V4.1-Flash on SVDB. These programs are the outputs of detector development and run without language-model calls. Their evaluation follows Appendix B.1.

(a) Qwen3.8-27B · SMAP — VUS-PR **0.587**

Excursion dwell time relative to the longest normal excursion seen in train — *selection 0.161 → 0.326*

```python
"""Train-profile persistence detector (v2) for level-structured multichannel series.

Learned from labeled-normal train segments (one-class):
  * per series/channel profile: robust scale, level set, mode level,
    longest "high-deviation episode" (consecutive samples with
    |v - global series median| > k * scale), per-level run p99 (discrete);
  * profile selection by distribution signature: (p2, p50, p98) quantile
    distance plus coarse level-set Jaccard, pooled fallback when no
    train profile is close;
  * per-channel score = max of
      - devdwell: episode length L vs norm = max(train_eps_max, 3),
        r = min(L/norm, 8), base r^2; episodes with r > 1.25 get the
        epidemic kernel r^2 * exp(-d/24) with d = max(a-t, t-b) (negative inside:
        long episodes get amplified cores, wide halos keep the labeled
        onset/offset margins above nominal-episode scores); the kernel
        requires L >= norm + 6 so short nominal episodes stay flat;
      - leveldwell (discrete channels, non-mode levels): run length vs
        train per-level run p99 (novel levels L/3), weight 0.5.
  * final = max over channels.

Rationale: train elevations have tightly bounded durations (e.g. exactly
15-sample excursions) while validation anomalies last 2-3x longer; quiet
channels (never high in train) use k=3.5 so any sustained excursion is
flagged. Global median context keeps long flat excursions fully marked
(rolling context would follow them). Train-derived statistics only.
"""
import numpy as np

K_DEFAULT = 5.5
K_QUIET = 3.5
HALO_W = 24
HALO_SIG = 24.0
R_CAP = 8.0
SCORE_CAP = 1e6
BAND_GAP = 3
CT_GAP = 12
QUIET_CAP = 1.0
BG_WIN = 511
FLANK_W = 24
FLANK_MARGIN = 3
WIN_W = 128
WIN_CAP = 1e6
DISCRETE_MAX_UNIQ = 64
LEVELDWELL_MAX_UNIQ = 8


def _ctx_roll(x, W):
    """Centered rolling median of odd width W with reflected boundaries."""
    T = len(x)
    W = max(3, int(W))
    if W % 2 == 0:
        W -= 1
    if T <= 2:
        return np.full(T, np.median(x))
    h = W // 2
    if T <= h:
        return np.full(T, np.median(x))
    vext = np.concatenate([x[:h][::-1], x, x[-h:][::-1]])
    return np.median(np.lib.stride_tricks.sliding_window_view(vext, W), axis=1)


def _robust_scale(v):
    med = float(np.median(v))
    mad = float(np.median(np.abs(v - med)))
    p01 = float(np.percentile(v, 1))
    p99 = float(np.percentile(v, 99))
    return max(1.4826 * mad, 0.05 * (p99 - p01), 1e-8 * max(1.0, abs(med)))


def _runs(mask):
    out = []
    i = 0
    m = len(mask)
    while i < m:
        if mask[i]:
            j = i + 1
            while j < m and mask[j]:
                j += 1
            out.append((i, j))
            i = j
        else:
```

Figure 11: **Detector program (a): Qwen3.8-27B on SMAP.** Complete delivered source, lines 1–81 of 482; continued on the next 5 pages.

(a) Qwen3.8-27B · SMAP (continued) *lines 82–162, part 2 of 6*

```
            i += 1
    return out


def _level_runs(v):
    d = np.concatenate(([True], v[1:] != v[:-1]))
    st = np.nonzero(d)[0]
    en = np.concatenate((st[1:], (len(v),)))
    return list(zip(st.tolist(), en.tolist()))


def _p99_map(runs):
    return {float(lv): max(1, int(np.percentile(ns, 99))) for lv, ns in runs.items()}


def _mode_of(runs):
    if not runs:
        return None
    tot = {lv: sum(ns) for lv, ns in runs.items()}
    return float(max(tot, key=tot.get))


def _sig(v):
    return tuple(float(q) for q in np.percentile(v, [2, 50, 98]))


def _min_roll(x, W):
    T = len(x)
    W = max(3, int(W))
    if W % 2 == 0:
        W -= 1
    if T <= 2:
        return np.full(T, x.min())
    h = W // 2
    if T <= h:
        return np.full(T, x.min())
    vext = np.concatenate([x[:h][::-1], x, x[-h:][::-1]])
    return np.min(np.lib.stride_tricks.sliding_window_view(vext, W), axis=1)


def _contrast_states(v, p25):
    """Per high-quantile band: (threshold, max merged train run of
    {v >= thr} & {rolling-min-511 < p25})."""
    out = []
    mn = _min_roll(v, BG_WIN)
    low = mn < p25
    for q in (20, 30, 40, 50, 60, 70, 80):
        thr = float(np.percentile(v, q))
        if thr <= p25:
            continue
        mx = 0
        for a, b in _runs_gap(low & (v >= thr), CT_GAP):
            mx = max(mx, b - a)
        out.append((thr, int(mx)))
    return out


def _fit_profile(v):
    T = len(v)
    p = {
        'scale': float(_robust_scale(v)),
        'levels': [float(z) for z in np.unique(np.round(v, 6))],
        'sig': _sig(v),
        'eps_max': 0,
        'quiet': True,
        'lvl_runs': {},
        'lvl_p99': None,
        'mode': None,
        'bands': [],
        'p25': None,
        'cstates': [],
        'pos_norm': 0,
        'win': None,
    }
    if T >= 8:
        med = float(np.median(v))
        h = np.abs(v - med) > K_DEFAULT * p['scale']
        mx = 0
        for a, b in _runs(h):
            mx = max(mx, b - a)
        p['eps_max'] = int(mx)
```

Figure 11: **Detector program (a), continued.** Lines 82–162 of 482.

(a) Qwen3.8-27B · SMAP (continued) *lines 163–243, part 3 of 6*

```
        p['quiet'] = (mx == 0)
    if T >= 8 and len(p['levels']) >= 3:
        p['bands'] = _band_norms(v, p)
    else:
        p['bands'] = []
    if T >= 8:
        p25 = float(np.percentile(v, 25))
        p['p25'] = p25
        p['cstates'] = _contrast_states(v, p25)
    else:
        p['p25'] = None
        p['cstates'] = []
        p['pos_norm'] = 0
    if T >= 2 and len(p['levels']) <= DISCRETE_MAX_UNIQ:
        runs = {}
        for a, b in _level_runs(v):
            runs.setdefault(round(float(v[a]), 6), []).append(b - a)
        p['lvl_runs'] = runs
        p['lvl_p99'] = _p99_map(runs)
        p['mode'] = _mode_of(runs)
    if T >= 2 * WIN_W:
        wm, ws = _win_stats(v, WIN_W)
        wmm = float(np.median(wm))
        wsm = float(np.median(ws))
        p['win'] = (
            wmm,
            max(1.4826 * float(np.median(np.abs(wm - wmm))),
                0.01 * (float(np.percentile(wm, 99)) - float(np.percentile(wm, 1))),
                1e-12),
            wsm,
            max(1.4826 * float(np.median(np.abs(ws - wsm))),
                0.01 * (float(np.percentile(ws, 99)) - float(np.percentile(ws, 1))),
                1e-12),
        )
    return p


def _merge_win(wins):
    ws = [w for w in wins if w is not None]
    if not ws:
        return None
    return (
        float(np.median([w[0] for w in ws])),
        max(w[1] for w in ws),
        float(np.median([w[2] for w in ws])),
        max(w[3] for w in ws),
    )


def _merge_profiles(cands):
    if len(cands) == 1:
        return cands[0]
    allruns = {}
    for q in cands:
        for lv, ns in q.get('lvl_runs', {}).items():
            allruns.setdefault(lv, []).extend(ns)
    vs = np.concatenate([np.full(3, s) for s in (q['sig'] for q in cands)])
    bands = {}
    for q in cands:
        for thr, nrm in q.get('bands', []):
            bands[thr] = max(bands.get(thr, 0), nrm)
    cstates = {}
    for q in cands:
        for thr, nrm in q.get('cstates', []):
            cstates[thr] = max(cstates.get(thr, 0), nrm)
    p25s = [q['p25'] for q in cands if q.get('p25') is not None]
    return {
        'scale': max(float(q['scale']) for q in cands),
        'levels': sorted({lv for q in cands for lv in q['levels']}),
        'sig': tuple(float(np.median(vs[i * 3:(i + 1) * 3])) for i in range(3)),
        'eps_max': max(int(q['eps_max']) for q in cands),
        'quiet': all(bool(q['quiet']) for q in cands),
        'lvl_runs': allruns,
        'lvl_p99': _p99_map(allruns) if allruns else None,
        'mode': _mode_of(allruns),
        'bands': sorted(bands.items()),
        'cstates': sorted(cstates.items()),
        'p25': max(p25s) if p25s else None,
        'pos_norm': max(int(q.get('pos_norm', 0)) for q in cands),
        'win': _merge_win([q.get('win') for q in cands]),
    }
```

Figure 11: **Detector program (a), continued.** Lines 163–243 of 482.

(a) Qwen3.8-27B · SMAP (continued) *lines 244–324, part 4 of 6*

```python

def _pooled_profile(ch, profs):
    m = _merge_profiles([q[ch] for q in profs])
    # band and cstate thresholds are series-specific percentiles;
    # cross-applying them across dissimilar series is unsafe, so the
    # pooled fallback carries neither (it keeps scale/eps_max/levels)
    m['cstates'] = []
    m['bands'] = []
    return m


def _win_stats(v, W):
    T = len(v)
    n = T - W + 1
    if n <= 0:
        return np.array([]), np.array([])
    c = np.ones(W) / W
    m = np.convolve(v, c, 'full')[W - 1:W - 1 + n]
    d = np.convolve(v * v, c, 'full')[W - 1:W - 1 + n] - m * m
    s = np.sqrt(np.clip(d, 0, None))
    return m, s


def fit(train):
    if isinstance(train, (list, tuple)):
        segs = [np.asarray(a, dtype=float) for a in train]
    else:
        segs = [np.asarray(train, dtype=float)]
    profs = []
    F = None
    for s in segs:
        if s.ndim == 1:
            s = s[:, None]
        if s.shape[0] == 0:
            continue
        if F is None:
            F = s.shape[1]
        profs.append([_fit_profile(s[:, c]) for c in range(s.shape[1])])
    if F is None:
        F = 1
        profs = [[_fit_profile(np.zeros(2))] for _ in range(1)]
    pooled = [_pooled_profile(c, profs) for c in range(F)]
    return {'F': int(F), 'series': profs, 'pooled': pooled}


def _match(ch, v, state):
    lvv = set(np.round(np.unique(v), 6).tolist())
    coarse = set(np.round(np.unique(v), 2).tolist())
    sg = _sig(v)
    best = None
    bestd = np.inf
    bestcand = []
    for prof in state['series']:
        lvs = set(prof[ch]['levels'])
        if not lvs:
            continue
        q = np.array(prof[ch]['sig'])
        d = float(np.abs(q - np.array(sg)).sum())
        inter = len(coarse & set(np.round(list(lvs), 2)))
        union = len(coarse | set(np.round(list(lvs), 2)))
        jac = inter / union if union else 0.0
        if d < bestd - 1e-12:
            bestd = d
            bestcand = [prof[ch]]
        elif abs(d - bestd) <= 1e-12 and bestd < np.inf:
            bestcand.append(prof[ch])
    if bestd <= 0.5 and len(lvv) <= DISCRETE_MAX_UNIQ and bestcand:
        m = _merge_profiles(bestcand)
        # keep the merged profile's level bookkeeping only when the channel
        # is genuinely discrete in the scored series
        if len(lvv) > 64:
            m['lvl_p99'] = None
        return m
    return state['pooled'][ch]


def _runs_gap(mask, gap):
    """Runs of True, merging runs separated by gaps <= gap."""
    base = _runs(mask)
    out = []
```

Figure 11: **Detector program (a), continued.** Lines 244–324 of 482.

(a) Qwen3.8-27B · SMAP (continued) *lines 325–405, part 5 of 6*

```
    for a, b in base:
        if out and a - out[-1][1] <= gap:
            out[-1] = (out[-1][0], b)
        else:
            out.append((a, b))
    return out


def _band_norms(v, prof):
    """Per quantile band: (threshold, max merged train run) for bands that
    are non-trivial in train (threshold strictly above the train minimum)."""
    out = []
    vmin = min(prof['levels']) if prof['levels'] else -np.inf
    for q in (25, 50, 75, 90, 95):
        thr = float(np.percentile(v, q))
        if thr <= vmin + 1e-12:
            continue
        mask = v >= thr
        mx = 0
        for a, b in _runs_gap(mask, BAND_GAP):
            mx = max(mx, b - a)
        if mx > 0:
            out.append((thr, max(mx, 3)))
    return out


def _score_channel(v, prof):
    T = len(v)
    if T < 8:
        return np.zeros(T)
    sc = np.zeros(T)
    k = K_QUIET if prof['quiet'] else K_DEFAULT
    med = float(np.median(v))
    h = np.abs(v - med) > k * prof['scale']
    norm = max(int(prof['eps_max']), 3)
    for a, b in _runs(h):
        L = b - a
        r = min(L / norm, R_CAP)
        s = r * r
        sc[a:b] = np.maximum(sc[a:b], s)
        if r > 1.25 and L >= norm + 6:
            d0 = max(0, a - HALO_W)
            d1 = min(T, b + HALO_W)
            idx = np.arange(d0, d1)
            d = np.maximum(a - idx, idx - b)
            sc[d0:d1] = np.maximum(sc[d0:d1], s * np.exp(-d / HALO_SIG))
    if prof['quiet']:
        # quiet channels: cap the deviation-dwell contribution; the
        # band/contrast mechanisms below keep the ranking signal
        sc = np.minimum(sc, QUIET_CAP)
    win = prof.get('win')
    if (win is not None and prof['quiet'] and len(prof['levels']) > 8
            and T >= 2 * WIN_W):
        wm, ws = _win_stats(v, WIN_W)
        z = np.maximum(
            np.abs(wm - win[0]) / win[1],
            np.abs(ws - win[2]) / win[3],
        )
        z = np.minimum(z, WIN_CAP)
        n = len(z)
        ext = np.zeros(T)
        ext[WIN_W // 2: WIN_W // 2 + n] = z
        sc = np.maximum(sc, ext)
    bands = prof.get('bands')
    if bands:
        for thr, bnorm in bands:
            mask = v >= thr
            for a, b in _runs_gap(mask, BAND_GAP):
                L = b - a
                r = min(L / bnorm, R_CAP)
                s = r * r
                sc[a:b] = np.maximum(sc[a:b], s)
                if r > 1.25 and L >= bnorm + 6:
                    d0 = max(0, a - HALO_W)
                    d1 = min(T, b + HALO_W)
                    idx = np.arange(d0, d1)
                    d = np.maximum(a - idx, idx - b)
                    sc[d0:d1] = np.maximum(sc[d0:d1], s * np.exp(-d / HALO_SIG))
    p25 = prof.get('p25')
    cstates = prof.get('cstates')
    if p25 is not None and cstates:
```

Figure 11: **Detector program (a), continued.** Lines 325–405 of 482.

(a) Qwen3.8-27B · SMAP (continued) *lines 406–482, part 6 of 6*

```
        mn = _min_roll(v, min(BG_WIN, T - 1 if T % 2 == 0 else T))
        low = mn < p25
        med255 = _ctx_roll(v, 255)
        dep = med255 < p25
        for thr, cnorm_raw in cstates:
            mstate = low & (v >= thr)
            if not mstate.any():
                continue
            for a, b in _runs_gap(mstate, CT_GAP):
                # require low-context flanks on both sides: the excursion
                # must return to the depressed state (an event), not be a
                # one-sided transition out of it (recovery)
                if a - FLANK_MARGIN <= 0 or b + FLANK_MARGIN >= T:
                    continue
                lz = v[max(0, a - FLANK_W):a - FLANK_MARGIN]
                rz = v[b + FLANK_MARGIN:min(T, b + FLANK_W)]
                if lz.min() >= p25 or rz.min() >= p25:
                    continue
                L = b - a
                if cnorm_raw == 0:
                    # state never observed in train: raw dwell length
                    sc[a:b] = np.maximum(sc[a:b], min(L, 30.0))
                    continue
                cnorm = max(cnorm_raw, 3)
                r = min(L / cnorm, R_CAP)
                s = r * r
                # amplify by the width of the depressed-context region
                # containing the run: an event persists while the whole
                # operating state is depressed; in train the local median
                # never drops below p25, so the factor stays 1 there
                wmed = 0
                if dep[a + (b - a) // 2]:
                    i0 = a
                    while i0 > 0 and dep[i0 - 1]:
                        i0 -= 1
                    i1 = b
                    while i1 < T and dep[i1]:
                        i1 += 1
                    wmed = min(max(i1 - i0, 1), 511)
                if wmed > 25:
                    s *= 1.0 + np.log10(wmed / 25.0)
                sc[a:b] = np.maximum(sc[a:b], s)
                if r > 1.1 and L > cnorm:
                    d0 = max(0, a - HALO_W)
                    d1 = min(T, b + HALO_W)
                    idx = np.arange(d0, d1)
                    d = np.maximum(a - idx, idx - b)
                    sc[d0:d1] = np.maximum(sc[d0:d1], s * np.exp(-d / HALO_SIG))
    p99map = prof.get('lvl_p99')
    if p99map is not None and len(np.unique(v)) <= LEVELDWELL_MAX_UNIQ:
        mode = prof.get('mode')
        for a, b in _level_runs(v):
            L = b - a
            lv = round(float(v[a]), 6)
            if mode is not None and abs(lv - mode) <= 1e-9:
                continue
            r = L / p99map[lv] if lv in p99map else L / 3.0
            sc[a:b] = np.maximum(sc[a:b], 0.5 * min(r, 6.0) ** 2)
    return np.minimum(sc, SCORE_CAP)


def score(series, state):
    x = np.asarray(series, dtype=float)
    if x.ndim == 1:
        x = x[:, None]
    T = x.shape[0]
    out = np.zeros(T)
    for c in range(min(int(state['F']), x.shape[1])):
        v = x[:, c]
        bad = ~np.isfinite(v)
        if bad.any():
            v = v.copy()
            good = ~bad
            v[bad] = float(np.median(v[good])) if good.any() else 0.0
        prof = _match(c, v, state)
        out = np.maximum(out, _score_channel(v, prof))
    return out
```

Figure 11: **Detector program (a), continued.** Lines 406–482 of 482.

(b) GLM-5.3-Flash · GECCO — VUS-PR **0.878**

Persistent level steps, with one gate for each false-alarm family — *selection 0.406 → 0.765*

```python
import numpy as np

"""Two-scale persistent-step detector with magnitude-modulated gating for
the GECCO sensor grid.

Evidence chain (labeled train rows 0-64480; disclosed val crops and
recorded commit feedback of 173_GECCO_id_1_Sensor_tr_16165_1st_16265):
- Events are sustained (10-253 pt) level steps or ramps that revert; the
  stable process channels carry the signal (c2 elevated in every
  disclosed val event, c3 largest in train).
- The two high-scale channels (train sigma > 20x median) participate in
  almost no train event but dominate false alarms (regime holds, slow
  wander, aftermath relaxation, spikes).
- False-alarm families and their discriminators:
  * 1-3 pt bursts/spikes -> killed by a 3-point persistence min-filter.
  * Sustained holds -> both 64-pt short windows agree with the current
    level -> min-of-sides geomean gate suppresses.
  * Aftermath relaxation after big events -> deviation DECAYS while the
    long-background deviation magnitude is only moderate; events sustain
    or grow their deviation. Gating by the persistent long-background
    magnitude (zp_top1/theta) keeps events (zp >= 4) and trims
    relaxation rows (zp 1-4).

Scoring:
- z_c = |x - median(trailing 512)| / max(1.4826*MAD, 1.5*train_sigma);
  expanding window for the first 512 rows.
- zp = 3-point rolling min of z (persistence).
- Base channels: exclude channels with train sigma > 20x median sigma.
- base = zp[anchor c3] + mean(top-3 zp over base channels).
- gate = min(gB, gC) / (min + 1), gB/gC = eps(0.5)-clamped geomean of
  top-3 z from trailing/leading 64-pt backgrounds.
- magnitude_factor = min(zp_top1 / 4.0, 1).
- score = base * gate * magnitude_factor.
- Refractory: within 160 rows after a score peak > 8, scale the score by
  s / (peak * 0.12) (clamped to 1): fast event interiors keep factor 1,
  slow post-event recovery ramps collapse. Measured on recorded
  full-context val scores: aftermath FP points 105 -> 10, event rows
  above threshold unchanged (76/76), aftermath max 3.77 -> 0.76.
"""

W_LONG = 512
W_SHORT = 64
FLOOR = 1.5
ANCHOR = 3
KAPPA = 1.0
PERSIST = 1
GATE_EPS = 0.5
GATE_ZTHETA = 4.0
REFR_C = 160
REFR_H = 8.0
REFR_F = 0.12
SCALE_RATIO = 20.0
MIN_HISTORY = 8


def _robust_sigma(col):
    med = np.median(col)
    mad = np.median(np.abs(col - med))
    if mad <= 0:
        return max(float(np.std(col)), 1e-9)
    return max(1.4826 * mad, 1e-9)


def fit(train):
    if isinstance(train, (list, tuple)):
        parts = [np.asarray(p, dtype=float) for p in train if len(p)]
        x = np.vstack(parts) if parts else np.zeros((1, 1))
    else:
```

Figure 12: **Detector program (b): GLM-5.3-Flash on GECCO.** Complete delivered source, lines 1–68 of 204; continued on the next 2 pages.

(b) GLM-5.3-Flash · GECCO (continued) *lines 69–136, part 2 of 3*

```
        x = np.asarray(train, dtype=float)
    if x.ndim == 1:
        x = x[:, None]
    nch = x.shape[1]
    sig = np.array([_robust_sigma(x[:, c]) for c in range(nch)])
    med = np.nanmedian(x, axis=0)
    anchor = min(int(ANCHOR), nch - 1)
    med_sig = float(np.median(sig))
    base_ch = [c for c in range(nch) if sig[c] <= SCALE_RATIO * med_sig]
    if not base_ch:
        base_ch = [anchor]
    if anchor not in base_ch:
        base_ch = sorted(set(base_ch) | {anchor})
    return {
        "w_long": int(W_LONG),
        "w_short": int(W_SHORT),
        "floor": float(FLOOR),
        "kappa": float(KAPPA),
        "persist": int(PERSIST),
        "gate_eps": float(GATE_EPS),
        "gate_ztheta": float(GATE_ZTHETA),
        "refr_c": int(REFR_C),
        "refr_h": float(REFR_H),
        "refr_f": float(REFR_F),
        "anchor": anchor,
        "min_history": int(MIN_HISTORY),
        "scale_ratio": float(SCALE_RATIO),
        "gsig": sig,
        "cmed": med,
        "channels": int(nch),
        "base_channels": np.array(sorted(base_ch), dtype=int),
    }


def _roll(a, w, step, lead):
    Tn, Cn = a.shape
    pad = np.full((w, Cn), np.nan)
    ap = np.vstack([a, pad]) if lead else np.vstack([pad, a])
    sw = np.lib.stride_tricks.sliding_window_view(ap, w, axis=0)[:Tn, :, ::step]
    med = np.nanmedian(sw, axis=2)
    mad = np.nanmedian(np.abs(sw - med[:, :, None]), axis=2)
    return med, mad


def _expanding(a, w, minh):
    Tn, Cn = a.shape
    med = np.full((Tn, Cn), np.nan)
    mad = np.full((Tn, Cn), np.nan)
    for t in range(Tn):
        lo = max(0, t - w)
        if t - lo >= minh:
            win = a[lo:t]
            med[t] = np.median(win, axis=0)
            mad[t] = np.median(np.abs(win - med[t]), axis=0)
    return med, mad


def _zbg(x, w, lead, state):
    step = max(1, w // 64)
    med, mad = _roll(x, w, step, lead)
    if not lead:
        T = x.shape[0]
        if T > w:
            mh, dh = _expanding(x[:w], w, state["min_history"])
            med = np.vstack([mh, med[w:]])
            mad = np.vstack([dh, mad[w:]])
        else:
            med, mad = _expanding(x, w, state["min_history"])
```

Figure 12: **Detector program (b), continued.** Lines 69–136 of 204.

(b) GLM-5.3-Flash · GECCO (continued) *lines 137–204, part 3 of 3*

```python
    scale = np.maximum(1.4826 * mad, state["floor"] * state["gsig"][None, :])
    with np.errstate(invalid="ignore"):
        z = np.abs(x - med) / scale
    return np.where(np.isfinite(z), z, 0.0)


def _minfilter(a, k):
    out = a.copy()
    for i in range(1, k + 1):
        out[i:] = np.minimum(out[i:], a[:-i])
    return out


def _g3(z, chans, eps):
    zz = np.maximum(np.sort(z[:, chans], axis=1)[:, -3:], eps)
    return np.exp(np.log(zz).mean(axis=1))


def _runmax_prev(a, c):
    """max over [t-c, t-1]; causal, excludes the current row."""
    pad = np.concatenate([np.full(c, a[0]), a[:-1]])
    sw = np.lib.stride_tricks.sliding_window_view(pad, c)
    return sw.max(axis=1)[: len(a)]


def _refractory(s, state):
    c = state["refr_c"]
    if len(s) == 0 or c <= 0:
        return s
    peak = _runmax_prev(s, min(c, len(s)))
    factor = np.where(
        peak > state["refr_h"],
        np.clip(s / np.maximum(peak * state["refr_f"], 1e-12), 0.0, 1.0),
        1.0,
    )
    return s * factor


def score(series, state):
    x = np.asarray(series, dtype=float)
    if x.ndim == 1:
        x = x[:, None]
    T = x.shape[0]
    if T == 0:
        return np.zeros(0)
    xr = np.where(np.isfinite(x), x, state["cmed"][None, :])
    chans = [int(c) for c in state["base_channels"]]
    zL = _zbg(xr, state["w_long"], False, state)
    zB = _zbg(xr, state["w_short"], False, state)
    zC = _zbg(xr, state["w_short"], True, state)
    mB = _g3(zB, chans, state["gate_eps"])
    mC = _g3(zC, chans, state["gate_eps"])
    mBC = np.minimum(mB, mC)
    gate = mBC / (mBC + state["kappa"])
    zp = _minfilter(zL, state["persist"])
    zps = np.sort(zp[:, chans], axis=1)
    top3 = zps[:, -3:].mean(axis=1)
    top1 = zps[:, -1]
    anchor = min(state["anchor"], x.shape[1] - 1)
    base = zp[:, anchor] + top3
    mag = np.minimum(top1 / state["gate_ztheta"], 1.0)
    s = base * gate * mag
    s = _refractory(s, state)
    return np.asarray(s, dtype=float)


def detect(series):
    raise NotImplementedError
```

Figure 12: **Detector program (b), continued.** Lines 137–204 of 204.

(c) GLM-5.3 · MITDB — VUS-PR **0.867**

Beat rhythm plus off-beat spikes away from the QRS complex — *selection 0.449 → 0.597*

```python
import numpy as np

AMP_GATE = 1.2          # ratio above the series' p99.9 that counts as an excursion
MORPH_GATE = 3.5        # waveform distance (MAD units) above which a beat is boosted
SPIKE_GATE = 9.0        # off-beat robust |z| above which a point counts as a spike
SPIKE_CLEAR = 15        # samples around a beat that belong to the QRS itself
AMP_FRACTION = 12.0     # rolling window = beat period / AMP_FRACTION
IRR_GATE = 0.18         # local RR irregularity gate
RATE_GATE = 0.30        # |log(local median RR / global median RR)| gate
MAX_CV = 0.35           # coefficient-of-variation above which a beat train looks implausible


def _smooth(s, k=5):
    if len(s) < k:
        return np.asarray(s, dtype=float).copy()
    kernel = np.ones(k) / k
    return np.convolve(s, kernel, mode="same")


def _local_median(v, win, stride=25):
    """Rolling median on a coarse grid, linearly interpolated."""
    n = len(v)
    if n == 0:
        return v.copy()
    half = max(1, win // 2)
    idxs = np.arange(0, n, max(1, stride))
    if idxs[-1] != n - 1:
        idxs = np.append(idxs, n - 1)
    meds = np.empty(len(idxs))
    for j, i in enumerate(idxs):
        meds[j] = np.median(v[max(0, i - half):min(n, i + half + 1)])
    return np.interp(np.arange(n), idxs, meds)


def _candidate_peaks(s, win):
    sm = _smooth(s)
    n = len(sm)
    if n == 0:
        return np.array([], dtype=int), sm
    med = np.median(sm)
    hi_q = np.percentile(sm, 99.5)
    rng = hi_q - med
    if not np.isfinite(rng) or rng <= 0:
        return np.array([], dtype=int), sm
    # Global level gate plus a local level gate: the local term keeps
    # slow level-wander regions from generating spurious beats.
    thr = np.maximum(med + 0.3 * rng,
                     _local_median(sm, win) + 0.3 * rng)
    above = sm > thr
    peaks = []
    i = 1
    while i < n - 1:
        if above[i] and sm[i] >= sm[i - 1] and sm[i] > sm[i + 1]:
            lo, hh = max(0, i - 10), min(n, i + 11)
            peaks.append(lo + int(np.argmax(s[lo:hh])))
            i += 20
        i += 1
    return np.array(peaks, dtype=int), sm


def _detect_beats(s, win):
    """Beat (R-peak) positions with refractory non-maximum suppression."""
    peaks, sm = _candidate_peaks(s, win)
    if len(peaks) < 3:
        return peaks
    med_rr = np.median(np.diff(peaks))
    if not np.isfinite(med_rr) or med_rr <= 1:
        return peaks
    radius = max(2, int(0.5 * med_rr))
    order = np.argsort(-sm[peaks])
    accepted = []
    for idx in order:
        p = peaks[idx]
        if all(abs(p - q) >= radius for q in accepted):
            accepted.append(p)
    return np.array(sorted(accepted), dtype=int)


def _detect_on_lead(s):
    """Beat train for one lead at its own period, or an empty array."""
    peaks = _detect_beats(s, int(min(300, len(s))))
    if len(peaks) < 3:
        return np.array([], dtype=int), float("inf")
```

Figure 13: **Detector program (c): GLM-5.3 on MITDB.** Complete delivered source, lines 1–83 of 329; continued on the next 3 pages.

(c) GLM-5.3 · MITDB (continued) *lines 84–166, part 2 of 4*

```python
    med_rr = float(np.median(np.diff(peaks)))
    if not np.isfinite(med_rr) or med_rr <= 0:
        return np.array([], dtype=int), float("inf")
    peaks = _detect_beats(s, int(med_rr))
    if len(peaks) < 3:
        return np.array([], dtype=int), float("inf")
    rr = np.diff(peaks)
    cv = float(np.std(rr) / np.mean(rr)) if np.mean(rr) > 0 else float("inf")
    return peaks, cv


def _select_beats(x):
    """Beat train over all channels and both polarities.

    Prefer the first regular train in (ch0, +), (ch0, -), (ch1, +), (ch1, -)
    order; if no lead is regular, fall back to the most regular one.
    """
    candidates = []
    for c in range(x.shape[1]):
        for si, sign in enumerate((1.0, -1.0)):
            peaks, cv = _detect_on_lead(sign * x[:, c])
            if len(peaks) >= 3 and np.isfinite(cv):
                candidates.append(((c, si), cv, peaks))
    if not candidates:
        return np.array([], dtype=int)
    regular = [cand for cand in candidates if cand[1] < MAX_CV]
    if regular:
        return regular[0][2]
    return min(candidates, key=lambda cand: (cand[1], -len(cand[2])))[2]


def _rhythm_features(peaks):
    """Per-beat rhythm score plus gated irregularity / rate-change boosts."""
    n = len(peaks)
    out = np.zeros(n)
    if n < 3:
        return out
    rr = np.diff(peaks)
    glob_med = float(np.median(rr))
    if not np.isfinite(glob_med) or glob_med <= 0:
        return out
    for i in range(n):
        idx = [j for j in range(max(0, i - 4), min(n - 1, i + 4))
               if j != i - 1 and j != i]
        vals = np.array([rr[j] for j in idx]) if idx else rr
        m = float(np.median(vals))
        if not np.isfinite(m) or m <= 0:
            m = glob_med
        r_pre = (peaks[i] - peaks[i - 1]) / m if i > 0 else 1.0
        r_post = (peaks[i + 1] - peaks[i]) / m if i < n - 1 else 1.0
        s = max(0.0, 1.0 - r_pre) + max(0.0, r_post - 1.0)
        # local irregularity (mean absolute RR deviation from local median)
        irr = float(np.mean(np.abs(vals - m))) / m
        # sustained local-vs-global rate change
        rate = abs(np.log(m / glob_med))
        s += max(0.0, irr - IRR_GATE)
        s += max(0.0, rate - RATE_GATE)
        out[i] = s
    return out


def _morphology_scores(x, peaks, med_rr):
    """Waveform distance of each beat to the median of its neighbours.

    Beats inside a labelled anomaly region keep their neighbours in the same
    series, so only a genuine local shape change produces a large value.
    """
    n = len(peaks)
    if n < 5:
        return np.zeros(n)
    a = max(6, int(round(0.14 * med_rr)))
    b = max(12, int(round(0.21 * med_rr)))
    T = x.shape[0]
    meds = []
    mads = []
    for c in range(x.shape[1]):
        v = np.asarray(x[:, c], dtype=float)
        med = np.median(v)
        mad = 1.4826 * np.median(np.abs(v - med)) + 1e-9
        meds.append(med)
        mads.append(mad)
    wins = []
    for p in peaks:
```

Figure 13: **Detector program (c), continued.** Lines 84–166 of 329.

(c) GLM-5.3 · MITDB (continued) *lines 167–249, part 3 of 4*

```
        lo, hi = p - a, p + b
        wins.append(x[lo:hi] if lo >= 0 and hi <= T else None)
    out = np.zeros(n)
    for i in range(n):
        if wins[i] is None:
            continue
        nb = []
        for j in range(max(0, i - 5), min(n, i + 6)):
            if abs(j - i) <= 1 or wins[j] is None:
                continue
            nb.append(wins[j])
        if len(nb) < 3:
            continue
        tmpl = np.median(np.array(nb), axis=0)
        acc = 0.0
        for c in range(x.shape[1]):
            d = wins[i][:, c] - tmpl[:, c]
            acc += float(np.mean(d * d)) / (mads[c] ** 2)
        out[i] = float(np.sqrt(acc / x.shape[1]))
    return out


def _off_beat_spike(x, peaks, clear):
    """Pointwise robust |z| at points away from any detected beat."""
    zs = []
    for c in range(x.shape[1]):
        v = np.asarray(x[:, c], dtype=float)
        med = np.median(v)
        mad = 1.4826 * np.median(np.abs(v - med)) + 1e-9
        zs.append(np.abs(v - med) / mad)
    z = np.max(np.array(zs), axis=0)
    if len(peaks) == 0:
        return z
    pos = np.arange(len(z))
    k = np.searchsorted(peaks, pos)
    k = np.clip(k, 1, len(peaks) - 1)
    dist = np.minimum(np.abs(pos - peaks[k - 1]), np.abs(pos - peaks[k]))
    z = np.where(dist > clear, z, 0.0)
    return z


def _rolling_mean(z, w):
    pad = w // 2
    zp = np.concatenate([np.full(pad, z[0]), z, np.full(w - 1 - pad, z[-1])])
    c = np.cumsum(np.concatenate([[0.0], zp]))
    return (c[w:] - c[:-w]) / w


def _amplitude_ratio(x, w):
    """Sustained-deviation profile: rolling mean of the per-channel robust |z|,
    normalized by a high quantile of its own distribution."""
    zs = []
    for c in range(x.shape[1]):
        v = np.asarray(x[:, c], dtype=float)
        med = np.median(v)
        mad = 1.4826 * np.median(np.abs(v - med)) + 1e-9
        zs.append(np.abs(v - med) / mad)
    z = np.max(np.array(zs), axis=0)
    rm = _rolling_mean(z, w)
    ref = np.percentile(rm, 99.9)
    if not np.isfinite(ref) or ref <= 0:
        return None
    return rm / ref


def _fallback_score(x):
    """Diff-based robust z-score when beat detection is not possible."""
    T = x.shape[0]
    scores = np.zeros(T)
    if T < 3:
        return scores
    for c in range(x.shape[1]):
        d = np.abs(np.diff(x[:, c]))
        med = np.median(d)
        mad = 1.4826 * np.median(np.abs(d - med)) + 1e-9
        z = np.abs(d - med) / mad
        scores = np.maximum(scores, np.concatenate(([0.0], z)))
    return scores


def fit(train):
    segs = train
    if isinstance(segs, (list, tuple)):
```

Figure 13: **Detector program (c), continued.** Lines 167–249 of 329.

(c) GLM-5.3 · MITDB (continued) *lines 250–329, part 4 of 4*

```
        segs = [np.asarray(s, dtype=float) for s in segs]
    else:
        segs = [np.asarray(segs, dtype=float)]
    scales = []
    for s in segs:
        if s.ndim == 2 and s.shape[0] > 0:
            scales.append(float(np.nanstd(s)))
    return {
        "n_segments": len(segs),
        "scale": float(np.median(scales)) if scales else 1.0,
    }


def score(series, state):
    x = np.asarray(series, dtype=float)
    if x.ndim == 1:
        x = x[:, None]
    T = x.shape[0]
    if T < 16:
        return np.zeros(T)
    peaks = _select_beats(x)
    if len(peaks) < 3:
        return _fallback_score(x)
    med_rr = float(np.median(np.diff(peaks)))
    if not np.isfinite(med_rr) or med_rr <= 0:
        return _fallback_score(x)

    n = len(peaks)
    off = int(round(med_rr / 2.0))
    bounds = []
    for i in range(n):
        s = max(0, peaks[i] - off)
        if i + 1 < n:
            e = peaks[i + 1] - off + 1
        else:
            e = peaks[i] - off + int(round(med_rr)) + 1
        bounds.append((s, min(e, T)))

    out = np.zeros(T)
    beat = _rhythm_features(peaks)
    for i in range(n):
        s, e = bounds[i]
        if e > s:
            out[s:e] = beat[i]

    # Off-beat spike component: excursions far from any detected beat.
    # Ordinary ECG never exceeds ~6.5 robust-z away from its QRS.
    off = _off_beat_spike(x, peaks, SPIKE_CLEAR)
    for i in range(n):
        s, e = bounds[i]
        if e > s:
            boost = max(0.0, float(off[s:e].max()) - SPIKE_GATE)
            if boost > 0:
                out[s:e] += boost

    # Waveform-shape component: catches beats whose local morphology is far
    # from the surrounding beats even when the rhythm is undisturbed.
    morph = _morphology_scores(x, peaks, med_rr)
    if morph is not None:
        for i in range(n):
            boost = max(0.0, float(morph[i]) - MORPH_GATE)
            if boost > 0:
                s, e = bounds[i]
                if e > s:
                    out[s:e] += boost

    # Sustained-amplitude component: catches wide extreme excursions that
    # rhythm alone misses.  Calibrated per series, gated so ordinary
    # beats (including ordinary R peaks) contribute nothing.
    if T >= 4000:
        w = int(np.clip(round(med_rr / AMP_FRACTION), 10, 100))
        ratio = _amplitude_ratio(x, w)
        if ratio is not None:
            for i in range(n):
                s, e = bounds[i]
                if e > s:
                    boost = max(0.0, float(ratio[s:e].max()) - AMP_GATE)
                    if boost > 0:
                        out[s:e] += boost
    return out
```

Figure 13: **Detector program (c), continued.** Lines 250–329 of 329.

(d) DeepSeek-V4.1-Flash · SVDB — VUS-PR **0.563**

Beat-synchronous novelty along morphology, amplitude, and interval axes — *selection 0.324 → 0.612*

```python
"""Beat-synchronous multi-axis novelty detector for quasi-periodic 2-channel series.

Train evidence (labels used only for offline feature study):
* Every labeled anomaly run spans roughly one cardiac cycle (median run 83-166 samples,
  i.e. ~0.6-1.4 x the dominant beat lag), so the scoring unit is one beat, not a window.
* Three per-beat axes separate labeled cycles from unlabeled ones and are partly
  complementary:
    - morphology novelty: 1 - max-shift normalized correlation of the beat's
      multi-channel epoch against the series' robust median beat template
      (labels sit at 0.11-1.07 vs ~0.01-0.42 for other cycles),
    - amplitude novelty: log ratio of the epoch's peak absolute deviation to the
      median over cycles (labeled cycles are systematically larger, +0.13..+1.38),
    - interval novelty: log ratio of the enclosing RR intervals to their medians
      (labeled cycles sit in locally longer/short cycle pairs).
  Adding the last two raised offline train macro-AUC from 0.85 to 0.92 and macro-AP
  from 0.25 to 0.37 over 20 labeled series, with 18/20 series improving.
* A fourth shape-radius axis: the spectral centroid of the shape-normalized epoch
  (mean frequency of |rfft| averaged over channels).  Labeled cycles have a lower
  centroid (wider / slower complex) in 17/20 train series; subtracting its
  standardized value improved offline train macro-AP in 19/20 series
  (0.371 -> 0.435) and macro-AUC in 14/20 (0.9209 -> 0.9214).
* Two further cheap per-beat axes carry independent information:  the epoch skewness
  of the shape-normalized multi-channel epoch (labeled cycles are more asymmetric;
  positive in 13/20 train series) and the number of envelope peaks above 0.4 of the
  epoch maximum (fragmented / multi-component complexes; positive in 18/20 series).
  Adding both with weights 0.3/0.3 raised offline train macro-AUC 0.9411 -> 0.9444
  and macro-AP 0.4750 -> 0.5006 (12/20 and 14/20 series better).
* Each axis is standardized self-referentially (median / MAD over that series' own
  cycles), so a fixed scale-free combination transfers across records.
* Beat localization matters: the summed-|derivative| envelope peaks on the QRS
  slope, not on the activation extremum, so epochs were misaligned by 10-30 samples.
  Snapping every detected beat to the strongest |amplitude| extremum within +-P*0.35
  (with a minimum P/3 separation) moved offline train macro-AUC 0.922 -> 0.944 and
  macro-AP 0.415 -> 0.477, improving 19/20 and 18/20 labeled series.
* A beat's combined score is spread as a plateau over its own cycle interval:
  0.5 x the previous RR back and 0.65 x the next RR forward.  Labeled windows sit
  slightly later than the detected activation, and the asymmetric plateau improved
  offline train macro-AP 0.501 -> 0.534 in 19/20 series while macro-AUC stayed at
  0.944 -> 0.947, whereas a symmetric 0.6/0.6 (or longer leads) was worse.

Nothing here is record specific: fit() stores only pooled preprocessing constants and
score() re-derives period, beats, template and robust scales from the series it is given.
"""
import numpy as np

__all__ = ["fit", "score"]

_WMULT = 0.75      # epoch half width as a fraction of the median RR
_DIL_LEAD = 0.5    # plateau reach back over the previous RR interval
_DIL_LAG = 0.65    # plateau reach forward over the next RR interval
_ALIGN = 8         # +/- sample shifts tried when matching the template
_SNAP = 0.35       # beat re-localization radius as a fraction of the estimated period
_W_AMP = 0.5       # weight of the amplitude axis
_W_CENT = 0.3      # weight of the (negative) spectral-centroid axis
_W_SKEW = 0.3      # weight of the epoch-skewness axis
_W_PK = 0.3        # weight of the epoch envelope-peak-count axis
_W_RR = 0.3        # weight of the interval axis
_ZCLIP = 8.0


def _finite(x):
    x = np.asarray(x, dtype=np.float64)
    if x.ndim == 1:
        x = x[:, None]
    if not np.isfinite(x).all():
        x = x.copy()
        col = np.nanmedian(np.where(np.isfinite(x), x, np.nan), axis=0)
        col = np.where(np.isfinite(col), col, 0.0)
        bad = np.flatnonzero(~np.isfinite(x).any(axis=1))
        if bad.size:
            x[bad] = col[None, :]
        x = np.where(np.isfinite(x), x, col[None, :])
    return np.ascontiguousarray(x, dtype=np.float64)


def _robust_norm(x):
    mu = np.median(x, axis=0)
    sd = 1.4826 * np.median(np.abs(x - mu), axis=0)
    sd = np.where(sd < 1e-9, x.std(axis=0) + 1e-9, sd)
    sd = np.where(sd < 1e-12, 1.0, sd)
    return (x - mu) / sd


def _envelope(xn, smooth=7):
    """Activation envelope (summed absolute derivative), length preserved."""
    T = xn.shape[0]
```

Figure 14: **Detector program (d): DeepSeek-V4.1-Flash on SVDB.** Complete delivered source, lines 1–86 of 343; continued on the next 3 pages.

(d) DeepSeek-V4.1-Flash · SVDB (continued) *lines 87–172, part 2 of 4*

```
    d = np.abs(np.diff(xn, axis=0)).sum(1)
    if d.size == 0:
        return np.zeros(T, dtype=np.float64)
    k = np.ones(int(smooth), dtype=np.float64) / float(smooth)
    e = np.convolve(d, k, mode="same")
    if e.size < T:
        tail = np.full(T - e.size, float(e[-1]) if e.size else 0.0)
        e = np.concatenate([e, tail])
    return e[:T]


def _period_acf(e, lo=20, hi=400, max_len=65536):
    """Dominant repeat lag from the FFT autocorrelation of the envelope."""
    e = np.asarray(e, dtype=np.float64)
    if e.size > max_len:
        e = e[:max_len]
    a = e - e.mean()
    if a.size < 8 or not np.isfinite(a).all() or float(np.abs(a).max()) <= 0:
        return 0, 0.0
    nfft = int(2 ** int(np.ceil(np.log2(2.0 * a.size))))
    f = np.fft.rfft(a, nfft)
    ac = np.fft.irfft(f * np.conj(f), nfft)[: a.size]
    if ac[0] <= 0:
        return 0, 0.0
    ac = ac / ac[0]
    hi = int(min(hi, ac.size - 2))
    lo = int(max(2, min(lo, hi - 1)))
    best, bestv = 0, 0.0
    for i in range(lo + 1, hi):
        if ac[i] > ac[i - 1] and ac[i] >= ac[i + 1] and ac[i] > 0.2 and ac[i] > bestv:
            best, bestv = i, float(ac[i])
    return int(best), float(bestv)


def _detect_beats(xn, period, frac=0.5, smooth=7):
    e = _envelope(xn, smooth=smooth)
    n = e.size
    if n < 8 or period < 4:
        return np.zeros(0, dtype=int), e
    cand = np.flatnonzero((e[1:-1] > e[:-2]) & (e[1:-1] >= e[2:])) + 1
    thr = np.percentile(e, 60.0)
    cand = cand[e[cand] > thr]
    if cand.size == 0:
        return np.zeros(0, dtype=int), e
    r = max(2, int(period // 6))
    peaks = []
    for c in cand:
        a, b = max(0, int(c) - r), min(n, int(c) + r + 1)
        peaks.append(a + int(np.argmax(e[a:b])))
    peaks = np.unique(np.asarray(peaks, dtype=int))
    order = peaks[np.argsort(-e[peaks])]
    half = max(2, int(period * frac))
    occ = np.zeros(n, dtype=bool)
    taken = []
    for i in order:
        a, b = max(0, i - half), min(n, i + half + 1)
        if not occ[a:b].any():
            taken.append(int(i))
            occ[a:b] = True
    return np.sort(np.asarray(taken, dtype=int)), e


def _snap_beats(xn, pk, period):
    """Re-localize each beat on the strongest amplitude extremum near its envelope peak.

    The envelope peak sits on an activation slope; the epoch template is built from the
    epochs themselves, so consistent alignment on the extremum sharpens template matching.
    A minimum separation of period/3 keeps neighbouring beats from collapsing onto one peak.
    """
    if pk.size == 0:
        return pk
    T = xn.shape[0]
    r = max(2, int(round(period * _SNAP)))
    min_sep = max(2, int(period / 3.0))
    mag = np.abs(xn).sum(axis=1)
    out = []
    for p in pk:
        a = max(0, int(p) - r)
        b = min(T, int(p) + r + 1)
        if b <= a:
            continue
        j = a + int(np.argmax(mag[a:b]))
        if not out or j > out[-1] + min_sep:
            out.append(int(j))
    if len(out) < 5:
        return pk
```

Figure 14: **Detector program (d), continued.** Lines 87–172 of 343.

(d) DeepSeek-V4.1-Flash · SVDB (continued) *lines 173–258, part 3 of 4*

```
    return np.asarray(out, dtype=int)


def _z(v, clip=_ZCLIP):
    v = np.asarray(v, dtype=np.float64)
    if v.size == 0:
        return v
    m = float(np.median(v))
    s = 1.4826 * float(np.median(np.abs(v - m)))
    if not np.isfinite(s) or s < 1e-9:
        s = float(v.std())
    if not np.isfinite(s) or s < 1e-12:
        s = 1.0
    return np.clip((v - m) / s, -clip, clip)


def fit(train):
    """Pooled, label-free preprocessing constants (robust channel scales + typical lag)."""
    if isinstance(train, (list, tuple)):
        segs = [np.asarray(s, dtype=np.float64) for s in train if np.size(s)]
    else:
        segs = [np.asarray(train, dtype=np.float64)]
    scales, periods = [], []
    for s in segs:
        if s.ndim == 1:
            s = s[:, None]
        if s.shape[0] < 64:
            continue
        s = _finite(s)
        mu = np.median(s, axis=0)
        sd = 1.4826 * np.median(np.abs(s - mu), axis=0)
        scales.append(np.where(sd < 1e-12, 1.0, sd))
        P, amp = _period_acf(_envelope(_robust_norm(s)))
        if P > 0 and amp > 0.25:
            periods.append(P)
    scale = np.ones(1, dtype=np.float64)
    if scales:
        L = min(len(s) for s in scales)
        M = np.stack([s[:L] for s in scales])
        scale = np.maximum(np.median(M, axis=0), 1e-12)
    period = float(np.median(periods)) if periods else 100.0
    return {"channel_scale": [float(v) for v in scale],
            "default_period": float(period),
            "n_channels": int(scale.size),
            "n_train_segments": int(len(segs))}


def score(series, state):
    x = _finite(series)
    T, F = x.shape
    out = np.zeros(T, dtype=np.float64)
    if T < 64:
        return out
    state = state or {}
    try:
        default_period = float(state.get("default_period", 100.0) or 100.0)
    except Exception:
        default_period = 100.0
    if not np.isfinite(default_period):
        default_period = 100.0

    xn = _robust_norm(x)
    e = _envelope(xn)
    period, amp_acf = _period_acf(e)
    if period <= 0 or amp_acf <= 0.25:
        period = int(max(16, min(400, round(default_period))))

    pk, e = _detect_beats(xn, period)
    pk = _snap_beats(xn, pk, period)
    nbeat = int(pk.size)
    if nbeat < 5:
        thr = np.percentile(e, 99.0)
        return np.clip(e - thr, 0.0, None)

    rr = np.diff(pk).astype(np.float64)
    ref = float(np.median(rr)) if rr.size else float(period)
    if not np.isfinite(ref) or ref < 4:
        ref = float(max(4, period))
    before = np.full(nbeat, ref, dtype=np.float64)
    after = np.full(nbeat, ref, dtype=np.float64)
    if rr.size:
        before[1:] = rr
        after[:-1] = rr

    # ---- axis 1: morphology novelty vs the series' own median beat template
    W = int(np.clip(round(ref * _WMULT), 24, 300))
```

Figure 14: **Detector program (d), continued.** Lines 173–258 of 343.

(d) DeepSeek-V4.1-Flash · SVDB (continued) *lines 259–343, part 4 of 4*

```
    W = max(8, min(W, max(8, T // 2)))
    eps = []
    valid = []
    for i, p in enumerate(pk):
        if p - W >= 0 and p + W < T:
            valid.append(i)
            eps.append(xn[p - W : p + W].reshape(-1))
    dist = np.zeros(nbeat, dtype=np.float64)
    amp = np.zeros(nbeat, dtype=np.float64)
    cent = np.zeros(nbeat, dtype=np.float64)
    skew = np.zeros(nbeat, dtype=np.float64)
    npkv = np.zeros(nbeat, dtype=np.float64)
    if len(eps) >= 5:
        E = np.stack(eps).astype(np.float64)
        ez = E - E.mean(axis=1, keepdims=True)
        ez = ez / (ez.std(axis=1, keepdims=True) + 1e-9)
        tm = np.median(ez, axis=0)
        tm = tm - tm.mean()
        tm = tm / (tm.std() + 1e-9)
        L = float(tm.size)
        best = np.full(ez.shape[0], -1.0)
        for sh in range(-_ALIGN, _ALIGN + 1):
            best = np.maximum(best, (ez @ np.roll(tm, sh)) / L)
        d = np.clip(1.0 - best, 0.0, 2.0)
        epmax = np.abs(E).max(axis=1)
        med = float(np.median(epmax))
        if not np.isfinite(med) or med <= 1e-12:
            med = 1e-12
        a = np.log(np.maximum(epmax, 1e-12) / med)
        # shape radius: spectral centroid of the per-epoch z-scored epoch
        E3 = ez.reshape(ez.shape[0], -1, F)
        sp = np.abs(np.fft.rfft(E3, axis=1)).mean(axis=2)
        tot = sp.sum(axis=1)
        tot = np.where(tot > 1e-12, tot, 1e-12)
        fr = np.fft.rfftfreq(E3.shape[1])
        if fr.size == sp.shape[1]:
            cent_v = (sp * fr[None, :]).sum(axis=1) / tot
        else:
            cent_v = np.zeros(sp.shape[0], dtype=np.float64)
        # epoch asymmetry and envelope fragmentation
        sk = ez.reshape(ez.shape[0], -1, F)
        sk = (sk ** 3).mean(axis=1).mean(axis=1)
        dE = np.abs(np.diff(E3, axis=1))
        env = dE.sum(axis=2)
        mx = env.max(axis=1)
        mx = np.where(mx > 0, mx, 1.0)
        above = (env[:, 1:-1] > env[:, :-2]) & (env[:, 1:-1] >= env[:, 2:]) & (
            env[:, 1:-1] > 0.4 * mx[:, None])
        npk = above.sum(axis=1).astype(np.float64)
        vi = np.asarray(valid, dtype=int)
        dist[vi] = d
        amp[vi] = a
        cent[vi] = cent_v
        skew[vi] = sk
        npkv[vi] = npk
        fill = float(np.median(d)) if d.size else 0.0
        inv = np.setdiff1d(np.arange(nbeat), np.asarray(valid, dtype=int))
        if inv.size:
            dist[inv] = fill
            amp[inv] = 0.0
    else:
        dist[:] = 0.0
        amp[:] = 0.0
        cent[:] = 0.0
        skew[:] = 0.0
        npkv[:] = 0.0

    # ---- axis 3: interval novelty (both enclosing cycles relative to their medians)
    mb = float(np.median(before)) if before.size else ref
    ma = float(np.median(after)) if after.size else ref
    mb = mb if np.isfinite(mb) and mb > 1e-9 else max(ref, 1.0)
    ma = ma if np.isfinite(ma) and ma > 1e-9 else max(ref, 1.0)
    rr_rel = (before / mb) * (after / ma)
    rr_log = np.log(np.maximum(rr_rel, 1e-12))

    # ---- combine standardized axes and spread over each beat's own cycle
    combo = (_z(dist) + _W_AMP * _z(np.maximum(amp, 0.0)) + _W_RR * _z(rr_log)
             - _W_CENT * _z(cent) + _W_SKEW * _z(skew) + _W_PK * _z(npkv))
    combo = np.nan_to_num(combo, nan=0.0, posinf=0.0, neginf=0.0)
    for i, p in enumerate(pk):
        a_i = max(0, int(p - _DIL_LEAD * before[i]))
        b_i = min(T, int(p + _DIL_LAG * after[i]) + 1)
        if b_i > a_i:
            np.maximum(out[a_i:b_i], combo[i], out=out[a_i:b_i])
    return np.nan_to_num(out, nan=0.0, posinf=0.0, neginf=0.0)
```

Figure 14: **Detector program (d), continued.** Lines 259–343 of 343.