AD-AGENT: A Multi-agent Framework for End-to-end Anomaly Detection

Tiankai Yang[1,*], Junjun Liu[1,*], Wingchun Siu[1,*], Jiahang Wang[1], Zhuangzhuang Qian[1], Chanjuan Song[1], Cheng Cheng[2], Xiyang Hu[3], Yue Zhao[1]
[1]University of Southern California, [2]Carnegie Mellon University, [3]Arizona State University
{tiankaiy, junjunl, siuw, jwang857, alexqian, songchan}@usc.edu,
ccheng2x@gmail.com, xiyanghu@asu.edu, yzhao010@usc.edu

Abstract

Anomaly detection (AD) is essential in areas such as fraud detection, network monitoring, and scientific research. However, the diversity of data modalities and the increasing number of specialized AD libraries pose challenges for non-expert users who lack in-depth library-specific knowledge and advanced programming skills. To tackle this, we present AD-AGENT, an LLM-driven multi-agent framework that turns natural-language instructions into fully executable AD pipelines. AD-AGENT coordinates specialized agents for intent parsing, data preparation, library and model selection, documentation mining, and iterative code generation and debugging. Using a shared short-term workspace and a long-term cache, the agents integrate popular AD libraries like PyOD, PyGOD, and TSLib into a unified workflow. Experiments demonstrate that AD-AGENT produces reliable scripts and recommends competitive models across libraries. The system is open-sourced to support further research and practical applications in AD.

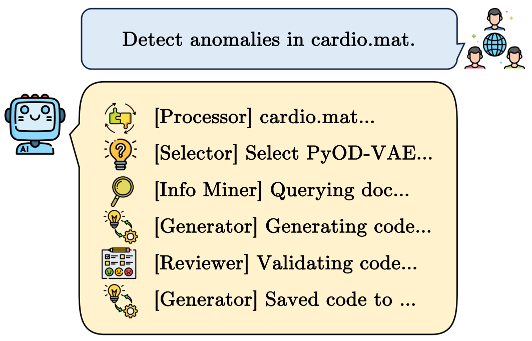
Detect anomalies in cardio.mat.
[Processor] cardio.mat...
[Selector] Select PyOD-VAE...
[Info Miner] Querying doc...
[Generator] Generating code...
[Reviewer] Validating code...
[Generator] Saved code to ...

Figure 1: Illustration of AD-AGENT: given a user request, the multi-agent system coordinates each stage to generate a runnable pipeline.

1 Introduction and Related Work

Anomaly detection (AD) plays a crucial role in a wide range of applications, including fraud detection (Abdallah et al., 2016), network monitoring (Sun et al., 2023), action recognition (Li et al., 2024b), and medical analysis (Fernando et al., 2021). To handle these diverse data types, the community has released modality-specific open-source libraries that package state-of-the-art models and utilities. Although these libraries accelerate experimentation, each introduces its own data formats and APIs, so users must "juggle" incompatible workflows before they can run even baseline methods. This learning overhead discourages adoption, especially among domain specialists who are not software/data engineers. The stakes are high: Knight Capital lost USD 440 million in 45 minutes when an unchecked trading anomaly cascaded through its systems (Heusser, 2012), and Target's 2013 breach has cost more than 200 million (U.S. Senate Committee on Commerce, Science, and Transportation, 2014). These incidents show that small gaps in an AD pipeline can cause major financial or security failures, showing the need for tooling that is both reliable and easy to integrate.

Meanwhile, large language models (LLMs) have demonstrated strong capabilities in reasoning (Guo et al., 2025), code generation (Liu et al., 2023), and tool use (Schick et al., 2023). Recent advances in agent-based systems have further enhanced the potential of LLMs to automate complex, multi-stage tasks that previously required substantial manual effort (Guan et al., 2023) (see extended related work in Appx. A). This presents a compelling opportunity: *Can we develop a general-purpose AD platform that leverages LLMs and existing libraries to build complete detection pipelines from the natural language intents of non-expert users?*

To address this, we introduce AD-AGENT– a multigent framework powered by LLMs that automates the construction of AD pipelines from plain language instructions. It decomposes the AD workflow into specialized agents responsible for user intent interpretation, data processing, library and

[*]Equal contribution.

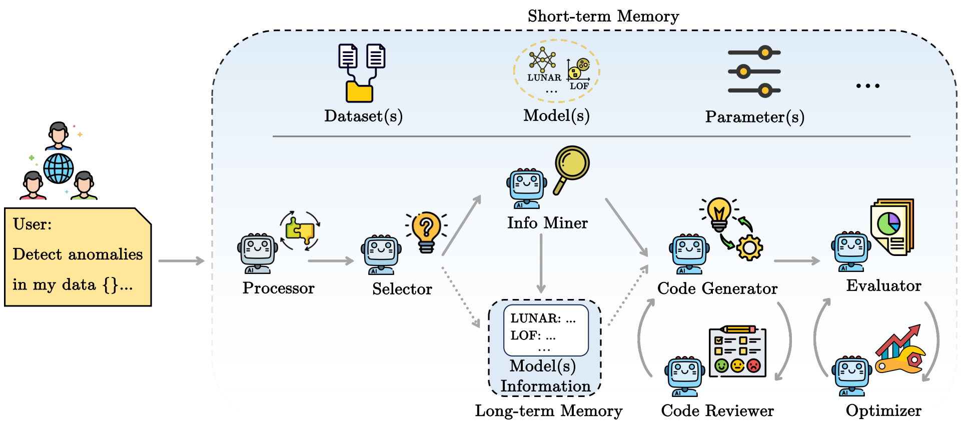

Figure 2: Flowchart of AD-AGENT. Users input natural language instructions and data from various modalities. AD-AGENT coordinates multiple LLM-powered agents via short-term and long-term memory to construct anomaly detection pipelines. Solid arrows represent the default workflow; dashed arrows indicate an optional path that bypasses web searches when algorithm information is stored in long-term memory.

model selection, knowledge retrieval, code generation and verification, and optional evaluation and tuning. For the memory mechanism, which is the key component to support agent-environment interactions (Zhang et al., 2024), we propose two memories. The short-term shared memory maintains the context of the current session, enabling coordination among agents, while the long-term memory serves as a cache to reduce costly queries across repeated sessions. By combining specialized agents with structured memory, AD-AGENT allows non-expert users to build comprehensive AD pipelines across multiple libraries and modalities using only natural language, relieving the need for library-specific expertise or manual programming. Figure 1 provides an illustration of AD-AGENT.

A survey of prior related LLM-agent work and modality-specific AD libraries is provided in Appendix A. **Our contributions are as follows**:

- **Unified multi-modal-library automation**. We propose the first multi-agent framework that integrates multiple domain-specific AD libraries, enabling end-to-end, cross-modality pipeline construction from natural language.
- **Modular, extensible, and long-lifecycle design**. Loosely coupled agents for reasoning, retrieval, and generation enable AD-AGENT to easily incorporate new libraries and tasks with minimal changes, supporting a long-lasting ecosystem.
- **Accessible to non-experts**. AD-AGENT converts natural language instructions into executable scripts and supports diverse data types, enabling non-expert users without programming skills or specialized knowledge to start easily.
- **Open-source release**. We release AD-AGENT at `https://github.com/USC-FORTIS/AD-AGENT` to provide the community with a practical, extensible platform for LLM-driven AD research and real-world applications.

2 Methodology

We present AD-AGENT, a multi-agent framework that automates AD across diverse modalities and use cases. By integrating established AD libraries – PyOD for multivariate data, PyGOD for graph data, and TSLib for time series – AD-AGENT supports a broad range of models and enables end-to-end automation from user instruction to script.

2.1 Agents

We decompose the detection workflow into multiple subtasks, with each stage handled by a specialized LLM-powered agent, as illustrated in Fig. 2.

Processor. Datasets in practice come in diverse formats (e.g., .csv, .mat, or even natural language), and detection tasks may vary from supervised setups to zero-shot scenarios. The Processor agent serves as the entry point of the system, using LLMs to interpret inputs, infer key attributes (e.g., modality, supervision type), and extract user-specified constraints. It organizes this information into a structured format that guides downstream agents.

Selector. Building on the Processor's output, the Selector agent determines which AD library best aligns with the inferred data modality and task requirements. If the user does not specify a model, the Selector recommends one from the chosen library. Inspired by recent advances in LLM-based model selection (Qin et al., 2025; Chen et al., 2024; Yang et al., 2024), it leverages the LLM's knowledge of models to provide context-aware suggestions tailored to the dataset and task.

Info Miner. Understanding how to apply a model often requires consulting multiple documentation sources, which can be time-consuming and challenging, especially for non-experts. The Info Miner agent performs this background research autonomously. It integrates "Web Search" function from OpenAI (OpenAI, 2025b) to learn from and summarize relevant documents, code examples, and online tutorials. The output includes model descriptions, instructions, and parameter definitions for later code generation.

Code Generator & Reviewer. These two agents collaborate to produce reliable detection scripts. The Generator composes code based on user instructions and knowledge from the Info Miner. To ensure correctness, the Reviewer validates the code through a dry run using LLM-generated synthetic samples, aiming to quickly catch any execution errors. If issues are detected, the two agents enter a feedback loop, iteratively refining the code until a valid and executable pipeline is achieved.

Evaluator & Optimizer. These two agents provide optional extensions for performance evaluation and hyperparameter tuning. The Evaluator runs the pipeline and summarizes detection results when ground truth labels are available for the target dataset. The Optimizer, inspired by Liu et al. (2025), performs LLM-powered hyperparameter tuning based on the provided training dataset. They operate in a feedback loop, iterating between parameter updates and performance assessment.

2.2 Agent Collaboration and Workflow

AD-AGENT facilitates collaboration through two memory structures: a shared **short-term memory** and a persistent **long-term memory**.

The short-term memory serves as the central workspace where agents read and write task-related content. It stores the user input, the processed dataset, selected models, and parameter configurations. This enables agents to operate independently while remaining context-aware.

The long-term memory caches model information retrieved by Info Miner. Since mining from web sources is often time-consuming and resource-intensive, the system first checks this cache for recent summaries before initiating a new web search. It is refreshed periodically (e.g., weekly), allowing the system to benefit from up-to-date resources while avoiding redundant queries.

Table 1: Pipeline generation performance by library, showing success rate (code runs without error), average latency, LLM token usage (input/output), and per-pipeline billing cost in US dollars. The time spent in Reviewer is related to the complexity of models, which explains the increase in TSLib.

Libraries	Success Rate (%)	Time (s)	In/Out Tokens	Cost (US $)
PyOD	100.0	24.0	3,272/667	0.015
PyGOD	91.1	19.6	3,143/673	0.015
TSLib	90.0	125.2	2,680/561	0.012

As shown in Fig. 2, the system begins with the Processor, which interprets the user's input and prepares the data. Based on this context, the Selector determines the appropriate library and, if unspecified by the user, recommends a suitable model. The Info Miner then gathers relevant model details, consulting either the long-term memory or the web. With this knowledge, the Code Generator and Reviewer collaboratively assemble and verify the detection pipeline iteratively until the code is valid. Users may then choose to enable the Evaluator and Optimizer for optional performance assessment and hyperparameter tuning.

This collaborative agent framework allows AD-AGENT to flexibly support multiple data types, including new libraries, adapt to varying input formats, and deliver usable outputs with minimal user effort. Each agent contributes a specialized capability, with LLMs enabling reasoning, adaptation, and coordination across the workflow.

3 Experiments

We evaluate AD-AGENT on reliability and efficiency in constructing executable AD pipelines from natural language instructions, the quality of model selection, and the effectiveness of long-term memory. See Appx. B.2 for the use case discussion and Appx. B.3 for improvements by Optimize.

Datasets and Models. We select datasets and models for each library from their corresponding benchmarks: Chen et al. (2024) for PyOD, Liu et al. (2022) for PyGOD, and Wu et al. (2023) for TSLib. See details in Appx. B.1.

3.1 Pipeline Generation

We first assess whether AD-AGENT can successfully generate runnable pipelines across datasets and models in each supported library. We use

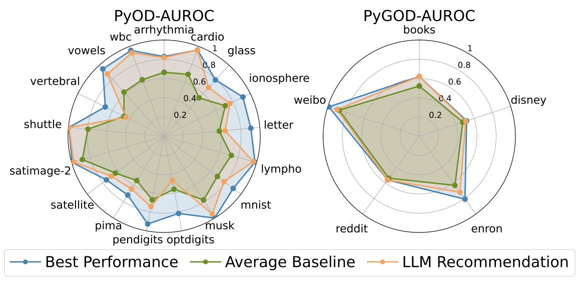

Figure 3: Model selection results for PyOD and PyGOD. We display the average AUROC of models recommended by querying the reasoning LLM three times (duplicates allowed). "Best Performance" marks the highest performance achieved by any available model for each dataset, while "Average Baseline" denotes the mean performance across all available models.

GPT-4o (OpenAI, 2024) to build all agents in our study. Table 1 presents the success rate, indicating whether the generated code runs without errors, the average generation time, and the average LLM token usage across different dataset–model pairs. We also use Llama 3.1 70B instruct (Dubey et al., 2024) as an open-source representative in Appx. B.5.

AD-AGENT demonstrates high reliability in producing valid pipelines across modalities, with low latency and manageable cost. We provide a complete example run in Appx. C for reference.

Correction Discussion. The feedback loop between the Code Generator and Reviewer often automatically corrects errors that occur during the initial code generation process. The most frequently fixed issues include missing or incorrectly assigned parameters and incorrect model import names. For example, when the Generator omits a required argument such as `n_features` for DeepSVDD, the Reviewer detects the resulting `TypeError`, references the correct constructor signature via the Info Miner, and amends the script accordingly. These correction cases demonstrate the practical benefit of the collaborative agent loop, allowing AD-AGENT to recover from common errors and increasing the pipeline success rate without user intervention.

Failure Discussion. While AD-AGENT demonstrates high overall reliability, a few recurring failure modes remain. Some failures arise from unaddressed internal data constraints. For instance, GAAN in PyGOD expects binary targets for its loss function, but the pipeline sometimes provides values outside the valid range. This highlights the need for improved data validation and type checking within both the Processor and Generator.

Additionally, some errors stem from library inconsistencies or incorrect functions, such as failed imports of DOMINAT in PyGOD, which is therefore excluded from the experiments, or input-size mismatches for Pyraformer in TSLib with certain datasets. While these are external, they underscore the need for AD-AGENT to integrate version checking and more robust fallback mechanisms.

Table 2: Average Web Search latency. Long-term memory lookups complete instantly and are omitted.

Libraries	PyOD	PyGOD	TSLib
Time (s)	10.6	12.0	10.8

3.2 Model Selection

We employ *o4-mini* (OpenAI, 2025a) to recommend AD models when the user leaves it unspecified. For each dataset, we query the LLM three times and compute the mean AUROC of selected models. Figure 3 compares the results in PyOD and PyGOD against two baselines: (*i*) the **best** result from any available model, indicating the upper performance limit; and (*ii*) the **average** performance of all available models, representing random selection. See more details and results in Appx. B.4.

The LLM's recommendations substantially exceed the average baseline and closely track the best performance in most datasets. This demonstrates that the Selector agent can harness LLM reasoning to choose proper models, simplifying model selection for non-expert users.

3.3 Long-term Memory Efficiency

To quantify the benefit of long-term memory, we compare the Info Miner's lookup latency and cost when using Web Search versus cached summaries. A typical Web Search takes about 10 seconds, as shown in Table 2, and costs 0.035 (US $) per call. In contrast, retrieving the same information from long-term memory is almost instantaneous and incurs no additional cost. This highlights the efficiency of long-term memory.

4 Conclusion

In this work, we introduced AD-AGENT, an LLM-powered multi-agent framework that automates end-to-end AD across multivariate, graph, and time-series data. By decomposing the workflow into specialized agents and coordinating them through short-term and long-term memory, AD-AGENT turns natural language instructions into runnable detection pipelines. Our experiments demonstrate high success rates of the system, accurate model recommendations, and substantial reductions in lookup latency and cost via long-term caching. The system is released for further research.

Future Directions. We plan to: *(i)* broaden AD-AGENT by continually adding new libraries and adapting other data modalities; *(ii)* support conversational interactions so users can iteratively refine pipelines; *(iii)* provide a secure, cloud-based workspace with pre-configured environments to simplify setup; *(iv)* introduce cost-aware planning that balances performance and LLM API budgets; and *(v)* envision a global, community-driven ecosystem where stakeholders collaborate on open-source tools for AD.

Limitations

Despite its flexibility and automation, AD-AGENT has several limitations. The system depends on the accuracy and currency of both the underlying LLMs and external libraries; breaking changes or undocumented features may lead to pipeline failures. Also, not all model or data-specific constraints can be automatically detected, which may result in occasional misconfigurations or runtime errors. Furthermore, AD-AGENT has been validated primarily on standard benchmarks, and its effectiveness and robustness for specialized or proprietary datasets need further systematic investigation.

Ethics Statement

This work adheres to established ethical standards in both research and software development. All experiments are conducted on public datasets, with no personally identifiable or sensitive information processed or disclosed. AD-AGENT is under the BSD 2-clause License, ensuring transparency and reproducibility. The system is designed to assist non-expert users in building AD pipelines. Additionally, ChatGPT was used exclusively to make minor grammatical improvements to the manuscript.

Social Impact Discussion

AD-Agent has the potential to lower barriers to anomaly detection by enabling non-expert users, including those in resource-constrained organizations, to prototype pipelines rapidly and researchers or educators to explore AD concepts through natural-language interfaces. This democratization can broaden the adoption of AD methods and accelerate practical applications. At the same time, we acknowledge potential risks: non-experts may misapply AD in safety-critical domains without understanding limitations, automation bias may reduce human oversight, and reliance on commercial APIs introduces cost and accessibility concerns. By making the system open-source and explicitly documenting limitations, we aim to encourage responsible use and enable the community to address these societal considerations.

Acknowledgments

This work was partially supported by the National Science Foundation under Award No. 2428039 and No. 2346158. We also acknowledge the use of computational resources provided by the Advanced Cyberinfrastructure Coordination Ecosystem (**?**): Services & Support (ACCESS) program, supported by NSF grants #2138259, #2138286, #2138307, #2137603, and #2138296. Specifically, this work used NCSA Delta GPU at the National Center for Supercomputing Applications (NCSA) through allocation CIS250073. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

Aisha Abdallah, Mohd Aizaini Maarof, and Anazida Zainal. 2016. Fraud detection system: A survey. *Journal of Network and Computer Applications*, 68:90–113.

Roel Bouman, Zaharah Bukhsh, and Tom Heskes. 2024. Unsupervised anomaly detection algorithms on real-world data: how many do we need? *Journal of Machine Learning Research*, 25(105):1–34.

Sihan Chen, Zhuangzhuang Qian, Wingchun Siu, Xingcan Hu, Jiaqi Li, Shawn Li, Yuehan Qin, Tiankai Yang, Zhuo Xiao, Wanghao Ye, and 1 others. 2024. Pyod 2: A python library for outlier detection with llm-powered model selection. *arXiv preprint arXiv:2412.12154*.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, and 1 others. 2024. The llama 3 herd of models. *arXiv e-prints*, pages arXiv–2407.

Tharindu Fernando, Harshala Gammulle, Simon Denman, Sridha Sridharan, and Clinton Fookes. 2021. Deep learning for medical anomaly detection–a survey. *ACM Computing Surveys (CSUR)*, 54(7):1–37.

Yile Gu, Yifan Xiong, Jonathan Mace, Yuting Jiang, Yigong Hu, Baris Kasikci, and Peng Cheng. 2025. Argos: Agentic time-series anomaly detection with autonomous rule generation via large language models. *arXiv preprint arXiv:2501.14170*.

Lin Guan, Karthik Valmeekam, Sarath Sreedharan, and Subbarao Kambhampati. 2023. Leveraging pre-trained large language models to construct and utilize world models for model-based task planning. *Advances in Neural Information Processing Systems*, 36:79081–79094.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, and 1 others. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*.

Taicheng Guo, Xiuying Chen, Yaqi Wang, Ruidi Chang, Shichao Pei, Nitesh V Chawla, Olaf Wiest, and Xiangliang Zhang. 2024. Large language model based multi-agents: a survey of progress and challenges. In *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence*, pages 8048–8057.

Songqiao Han, Xiyang Hu, Hailiang Huang, Minqi Jiang, and Yue Zhao. 2022. Adbench: Anomaly detection benchmark. *Advances in neural information processing systems*, 35:32142–32159.

Matthew Heusser. 2012. Software testing lessons learned from knight capital fiasco. *CIO Magazine*. Accessed: 2025-05-16.

Mahnaz Koupaee, Jake W Vincent, Saab Mansour, Igor Shalyminov, Han He, Hwanjun Song, Raphael Shu, Jianfeng He, Yi Nian, Amy Wing-mei Wong, and 1 others. 2025. Faithful, unfaithful or ambiguous? multi-agent debate with initial stance for summary evaluation. *arXiv preprint arXiv:2502.08514*.

Guohao Li, Hasan Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. 2023. Camel: Communicative agents for" mind" exploration of large language model society. *Advances in Neural Information Processing Systems*, 36:51991–52008.

Lincan Li, Jiaqi Li, Catherine Chen, Fred Gui, Hongjia Yang, Chenxiao Yu, Zhengguang Wang, Jianing Cai, Junlong Aaron Zhou, Bolin Shen, and 1 others. 2024a. Political-llm: Large language models in political science. *arXiv preprint arXiv:2412.06864*.

Shawn Li, Huixian Gong, Hao Dong, Tiankai Yang, Zhengzhong Tu, and Yue Zhao. 2024b. Dpu: Dynamic prototype updating for multimodal out-of-distribution detection. *arXiv preprint arXiv:2411.08227*.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems*, 36:21558–21572.

Junwei Liu, Kaixin Wang, Yixuan Chen, Xin Peng, Zhenpeng Chen, Lingming Zhang, and Yiling Lou. 2024a. Large language model-based agents for software engineering: A survey. *CoRR*.

Kay Liu, Yingtong Dou, Xueying Ding, Xiyang Hu, Ruitong Zhang, Hao Peng, Lichao Sun, and Philip S Yu. 2024b. Pygod: A python library for graph outlier detection. *Journal of Machine Learning Research*, 25(141):1–9.

Kay Liu, Yingtong Dou, Yue Zhao, Xueying Ding, Xiyang Hu, Ruitong Zhang, Kaize Ding, Canyu Chen, Hao Peng, Kai Shu, and 1 others. 2022. Bond: Benchmarking unsupervised outlier node detection on static attributed graphs. *Advances in Neural Information Processing Systems*, 35:27021–27035.

Siyi Liu, Chen Gao, and Yong Li. 2025. AgentHPO: Large language model agent for hyper-parameter optimization. In *The Second Conference on Parsimony and Learning (Proceedings Track)*.

Sizhe Liu, Yizhou Lu, Siyu Chen, Xiyang Hu, Jieyu Zhao, Yingzhou Lu, and Yue Zhao. 2024c. Drugagent: Automating ai-aided drug discovery programming through llm multi-agent collaboration. *arXiv preprint arXiv:2411.15692*.

OpenAI. 2024. Gpt-4o system card.

OpenAI. 2025a. Introducing openai o3 and o4-mini.

OpenAI. 2025b. New tools for building agents.

Yuehan Qin, Yichi Zhang, Yi Nian, Xueying Ding, and Yue Zhao. 2025. MetaOOD: Automatic selection of OOD detection models. In *The Thirteenth International Conference on Learning Representations*.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems*, 36:68539–68551.

Haiyang SHEN, Yue Li, Desong Meng, Dongqi Cai, Sheng Qi, Li Zhang, Mengwei Xu, and Yun Ma. 2025. Shortcutsbench: A large-scale real-world benchmark for API-based agents. In *The Thirteenth International Conference on Learning Representations*.

Chengyu Song, Linru Ma, Jianming Zheng, Jinzhi Liao, Hongyu Kuang, and Lin Yang. 2024. Audit-llm: Multi-agent collaboration for log-based insider threat detection. *arXiv preprint arXiv:2408.08902*.

Yongqian Sun, Daguo Cheng, Tiankai Yang, Yuhe Ji, Shenglin Zhang, Man Zhu, Xiao Xiong, Qiliang Fan, Minghan Liang, Dan Pei, and 1 others. 2023. Efficient and robust kpi outlier detection for large-scale datacenters. *IEEE Transactions on Computers*, 72(10):2858–2871.

U.S. Senate Committee on Commerce, Science, and Transportation. 2014. A “kill chain” analysis of the 2013 target data breach. Technical report, Majority Staff Report. Accessed: 2025-05-16.

Yuxuan Wang, Haixu Wu, Jiaxiang Dong, Yong Liu, Mingsheng Long, and Jianmin Wang. 2024. Deep time series models: A comprehensive survey and benchmark. *arXiv preprint arXiv:2407.13278*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems 35 (NeurIPS 2022)*, pages 24830–24843.

Haixu Wu, Tengge Hu, Yong Liu, Hang Zhou, Jianmin Wang, and Mingsheng Long. 2023. Timesnet: Temporal 2d-variation modeling for general time series analysis. In *The Eleventh International Conference on Learning Representations*.

Tiankai Yang, Yi Nian, Shawn Li, Ruiyao Xu, Yuangang Li, Jiaqi Li, Zhuo Xiao, Xiyang Hu, Ryan Rossi, Kaize Ding, and 1 others. 2024. Ad-llm: Benchmarking large language models for anomaly detection. *arXiv preprint arXiv:2412.11142*.

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

Zeyu Zhang, Xiaohe Bo, Chen Ma, Rui Li, Xu Chen, Quanyu Dai, Jieming Zhu, Zhenhua Dong, and Ji-Rong Wen. 2024. A survey on the memory mechanism of large language model based agents. *arXiv preprint arXiv:2404.13501*.

Appendix: AD-AGENT: A Multi-agent Framework for End-to-end Anomaly Detection

A Related Works

LLM-based multi-agent systems have emerged as a powerful paradigm for solving complex tasks through role specialization, planning, and tool use (Guo et al., 2024; Li et al., 2023).

These systems have been successfully applied to domains such as software engineering (Liu et al., 2024a), scientific discovery (Liu et al., 2024c), faithfulness evaluation (Koupaee et al., 2025), and social simulations (Li et al., 2024a). In the context of AD, Audit-LLM (Song et al., 2024) targets insider threat detection through multi-agent coordination, and Argos (Gu et al., 2025) uses LLM agents to generate interpretable anomaly rules for time-series monitoring. While effective, these systems are domain-specific and fixed in scope.

In parallel, several open-source libraries have been developed across different data modalities. Popular libraries such as PyOD (Chen et al., 2024), PyGOD (Liu et al., 2024b), and TSLib (Wang et al., 2024) provide strong support for AD on multivariate, graph, and time series data, respectively. While each library is effective within its domain, they differ in requirements and design. These inconsistencies make integration across libraries non-trivial.

AD-AGENT unifies multiple AD libraries within an LLM-driven multi-agent framework.

B Experiments Details

B.1 Datasets and Models

As mentioned in § 3, we adopt datasets and models for each library from corresponding benchmarks.

B.1.1 PyOD

Following PyOD 2 (Chen et al., 2024), we evaluated AD-AGENT on 17 widely used datasets originally from ADBench (Han et al., 2022), including arrhythmia, cardio, glass, ionosphere, letter, lympho, mnist, musk, optdigits, pendigits, pima, satellite, satimage-2, shuttle, vertebral, vowels, and WBC. For each dataset, we consider 10 models: ALAD, AnoGAN, AE, AE1SVM, DeepSVDD, DevNet, LUNAR, MO-GAAL, SO-GAAL, and VAE. See more details in Chen et al. (2024).

B.1.2 PyGOD

Following PyGOD (Liu et al., 2024b), we evaluated AD-AGENT on 5 real datasets originally from BOND (Liu et al., 2022), including books, disney, enron, reddit, weibo. For each dataset, we consider 9 models: AdONE, ANOMALOUS, AnomalyDAE, CONAD, DONE, GAAN, GUIDE, Radar, and SCAN. See more details in Liu et al. (2022).

Table 3: Detection Performance before and after executing Optimizer. Better results are highlighted in **bold**.

Models	AUROC(before→ after)	AUPRC(before→ after)
AE	0.7875 → **0.8732**	0.4191 → **0.4959**
ALAD	0.5861 → **0.6103**	0.1454 → **0.1624**
AnoGAN	0.8820 → **0.9438**	0.6050 → **0.7034**
AE1SVM	0.9450 → **0.9779**	0.6748 → **0.8388**
DeepSVDD	0.9259 → **0.9757**	0.6370 → **0.8046**
DevNet	0.0323 → 0.0323	0.0585 → 0.0585
LUNAR	0.5254 → **0.7941**	0.1736 → **0.4462**
MO-GAAL	0.5300 → **0.6200**	0.1900 → **0.2000**
SO-GAAL	0.6687 → **0.7724**	0.3512 → **0.4283**
VAE	0.9800 → 0.9800	0.8300 → 0.8300

B.1.3 TSLib

Wu et al. (2023) presents a benchmark study for TSLib (Wang et al., 2024). Following their approach, we evaluated AD-AGENT on 5 real-world datasets from Wu et al. (2023), including MSL, PSM, SMAP, SMD, and SWaT. For each dataset, we consider 10 models: Autoformer, DLinear, ETSformer, FEDformer, Informer, LightTS, Pyraformer, Reformer, TimesNet, and Transformer. See more details in Wu et al. (2023).

B.2 Use Cases Discussion

Our framework supports two common use cases frequently encountered in academic research and real-world deployments.

In research or benchmarking settings, users usually have access to a train/test split and ground-truth anomaly labels for the test set. AD-AGENT ingests the training data, builds the model, and reports metrics such as AUROC or F1 on the held-out test set if the user enables the Evaluator. Then the Optimizer can further refine hyperparameters by running an inner loop on the training data and passing a possibly better configuration back to the main pipeline before the final evaluation. This mirrors the evaluation protocol adopted by major AD benchmarks such as ADBench (Han et al., 2022).

In many production scenarios, only one raw, unlabeled dataset is available, and the goal is to identify anomalies directly within this set (Bouman

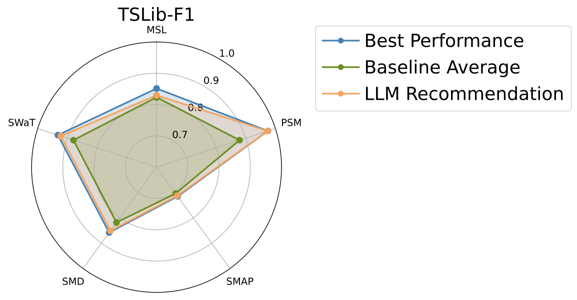

Figure 4: Model selection results for TSLib. We display the average F1-score of models recommended by querying the reasoning LLM three times (duplicates allowed). "Best Performance" marks the highest performance achieved by any available model for each dataset, while "Average Baseline" denotes the mean performance across all available models.

et al., 2024). In this case, AD-AGENT detects anomalies on the provided data in a single pass; the Evaluator and Optimizer remain inactive unless the user later supplies labels or a separate tuning set.

B.3 Optimizer Improvement

To demonstrate the impact of Optimizer, we evaluated it on the dataset "cardio" within PyOD. As shown in Table 3, Optimizer consistently improved detection quality. These results indicate that the Optimizer agent can automatically refine hyperparameters to produce significantly stronger AD pipelines without human intervention.

B.4 Additional Result of Model Selection

Figure 4 shows the model selection results in TSLib. LLM recommendation outperforms the average baseline in all datasets.

B.5 Open-source LLM Results

We select Llama 3.1 70B instruct (Dubey et al., 2024) as a representative of open-source LLMs. However, the performance is not promising: (i) open-source LLMs struggle to follow complex commands accurately, often producing inconsistent JSON formatting and failing to execute arguments correctly. (ii) Their limited abilities also hinder them from correcting invalid code in Reviewer.

Open-source or smaller LLM-based agents still lag behind closed-source or larger models in handling complex tasks (SHEN et al., 2025). This is a universal challenge for LLM agents. However, AD-AGENT is designed to ensure that even when using a powerful LLM like GPT-4o, the cost remains highly affordable.

C Example Run

Table 4 presents an actual session of AD-AGENT. In this example, a user requests to run VAE on the "cardio.mat" dataset via a simple natural language command: "Run VAE on cardio.mat." The system interprets the user's intent, processes the data, selects the appropriate library, retrieves model information, and automatically generates a runnable Python script. This example demonstrates the seamless collaboration between agents in AD-AGENT, showing how a single natural language instruction can be transformed into a ready-to-run AD pipeline with minimal user effort.

D Prompt Summary

AD-AGENT drives training-free LLM agents purely through carefully crafted prompts. Each prompt specifies the sub-task, enforces JSON blocks for inputs and outputs, and restricts the agent to a library-specific API surface, ensuring that downstream modules can trust the response without additional validation. Across all agents, we employ three core prompt engineering techniques: *chain-of-thought reasoning* to elicit step-by-step planning (Wei et al., 2022), strict *JSON-formatted outputs* for deterministic parsing, and a *self-revision loop* in which the Reviewer injects error traces into the LLM to correct code. The tables tables 5 to 9 present the raw prompts that implement these patterns for each agent.

```
=== [Main] Starting full pipeline ===
=== [Processor] Processing user input ===
Enter command (e.g., ‘Run IForest on glass_train.mat and glass_test.mat’):
User: Run VAE on cardio.mat
Experiment Configuration:
   Algorithm: [‘VAE’]
   Training Dataset: ./data/cardio.mat
   Testing Dataset:
   Parameters:
=== [Processor] User input processing complete

=== [Selector] Processing user input ===
=== [Selector] Selecting package & algorithm ===
Package name: pyod
{more content is omitted...}
=== [Selector] Selection complete ===

=== [Info Miner] Querying documentation for VAE ===
The Variational Autoencoder (VAE) in PyOD is {more content is omitted...}
**Initialization Function (‘__init__’):**
The ‘__init__’ method initializes the VAE model with {more content is omitted...}
**Parameters:**
- ‘contamination’:
   - **Type**: float in (0., 0.5)
   - **Default**: 0.1
   - **Description**: The proportion of outliers{more content is omitted...}
- ‘encoder_neuron_list’:
{more content is omitted...}
**Attributes:**
- ‘model’:
   - **Type**: torch.nn.Module
   - **Description**: The underlying VAE model.
{more content is omitted...}
**Python Dictionary of ‘__init__’ Parameters with Default Values:**
{
   ‘contamination”: 0.1,
   “encoder_neuron_list”: [128, 64, 32],
   {more content is omitted...}
}
[Cache Updated] Stored new documentation for VAE
=== [Info Miner] Documentation retrieved for VAE ===

=== [Code Generator] Generating code for VAE ===
=== [Code Reviewer] Validating for VAE ===
=== [Code Reviewer] Validation completed for VAE ===
=== [Code Generator] Saved code to ./generated_scripts/VAE_cardio.py ===
{more content is omitted...}
```

Table 4: A real example of AD-AGENT. The user provides a single natural language instruction (highlighted in green), and the system automatically parses the command, retrieves model metadata, and generates an executable Python script. Portions of the printed text are omitted (`{more content is omitted...}`) for brevity.

```
=== [Processor] Extraction prompt ===
Extract the algorithm, dataset_train, dataset_test, and optional parameters
from the above conversation
and return them in Python dictionary (JSON) format.
If any item is missing, return an empty object.
User input follows format ‘Run XXX on TRAIN_DATA and TEST_DATA with XXX’ where
‘with XXX’ and ‘TEST_DATA’ are optional.

For example: if the user says ‘Run IForest on ./data/train.mat and
./data/test.mat with contamination=0.1’
you should return
{‘algorithm’: [‘IForest’], ‘dataset_train’: ‘./data/train.mat’, ‘dataset_test’:
‘./data/test.mat’, ‘parameters’: {‘contamination’: 0.1} }

If user says ‘Run IForest on ./data/train.mat and ./data/test.mat’
you should return
{‘algorithm’: [‘IForest’], ‘dataset_train’: ‘./data/train.mat’, ‘dataset_test’:
‘./data/test.mat’, ‘parameters’: {} }

If user says ‘Run IForest’
you should return
{‘algorithm’: [‘IForest’], ‘dataset_train’: None, ‘dataset_test’: None,
‘parameters’: {} }

If user says ‘./data/train.mat and ./data/test.mat’
you should return
{‘algorithm’: [], ‘dataset_train’: ‘./data/train.mat’, ‘dataset_test’:
‘./data/test.mat’, ‘parameters’: {} }

IMPORTANT: DO NOT ASSUME ALGORITHM NAME OR PARAMETERS NAME.
IMPORTANT: Algorithm should always be an array.
IMPORTANT: IF USER WANTS TO RUN ALL ALGORITHMS, return ‘algorithm’ as [‘all’].
```

Table 5: Raw extraction prompt used by the **Processor** agent. The prompt instructs the LLM to extract algorithm names, dataset paths, and optional parameters from free-form user inputs, returning a structured Python dictionary.

```
=== [InfoMiner] Unified web-search prompt ===
You are a machine-learning expert and will assist me with researching a specific
use
of a deep-learning model in ‘{library_name}’.
Here is the official document you should refer to:
  ‘{doc_url}’
I want to run ‘{algorithm_name}’.  What is the initialisation function, its
parameters,
and its attributes? Briefly return the relevant documentation content.
Then extract ***all parameters*** of the __init__ method for the
‘{algorithm_name}’ class, along with their default values if available, and
return
a valid Python dictionary string in the following format:
{
   ‘param1’: default_value1,
   ‘param2’: default_value2,
   ...
}
If any default value is an object or function (e.g. ‘MinMaxScaler()’), wrap it
in quotes so the string remains valid for ast.literal_eval.
```

Table 6: Raw prompt template used by the **InfoMiner** agent. At runtime, the placeholders `{library_name}` and `{doc_url}` are filled according to their official documentation.

```
=== [CodeGen] PyOD labeled prompt (raw) ===
You are an expert Python developer with deep experience in anomaly detection libraries. Your
task is to:

1. Use the provided official documentation content for {algorithm} to understand how to use the
specified algorithm class, including initialization, training, and prediction methods.
2. Write only executable Python code for anomaly detection using PyOD and do not include any
explanations or descriptions.
3. Base your code strictly on the following official documentation excerpt:

– BEGIN DOCUMENTATION –
{algorithm_doc}
– END DOCUMENTATION –

4. The code should:
(1) import sys, os and include command ‘sys.path.append(os.path.abspath(os.path.join(os.path.
dirname(__file__), ‘..’)))’ in the head
(2) import DataLoader using following command ‘from data_loader.data_loader import DataLoader’
after (1)
(3) Initialize DataLoader using statement
    dataloader_train = DataLoader(filepath = {data_path_train}, store_script=True, store_path =
‘train_data_loader.py’)
    dataloader_test = DataLoader(filepath = {data_path_test}, store_script=True, store_path =
‘test_data_loader.py’)
(4) Use the statement
    X_train, y_train = dataloader_train.load_data(split_data=False)
    X_test, y_test = dataloader_train.load_data(split_data=False)
to generate variables X_train, y_train, X_test, y_test;
(5) Initialize the specified algorithm {algorithm} using variable ‘model’, strictly following
the provided documentation and train the model with X_train
(6) Determine whether the following parameters {parameters} apply to this initialization
function and, if so, add their values to the function.
(7) Use ‘.decision_scores_’ on X_train for training outlier scores
    Use ‘.decision_function(X_test)’ for test outlier scores
    Calculate AUROC (Area Under the Receiver Operating Characteristic Curve) and AUPRC (Area
Under the Precision-Recall Curve) based on given data
(8) Using variables to record the AUROC & AUPRC and print them out in following format:
    AUROC: \s*(\d+.\d+)
    AUPRC: \s*(\d+.\d+)
(9) Using variables to record prediction failed data and print these points out with true label
in following format:
    ‘Failed prediction at point [xx,xx,xx...] with true label xx’ Use ‘.tolist()’ to convert
point to be an array.

IMPORTANT:
- Strictly follow steps (2)-(8) to load the data from {data_path_train} & {data_path_test}.
- Do NOT input optional or incorrect parameters.
```

Table 7: Raw prompt used by the **CodeGenerator** agent for PyOD (labeled setting). The LLM generates executable code for training and evaluating anomaly detectors using the given documentation and dataset paths.

```
=== [Reviewer] Unit-test prompt ===
You will receive a Python script for {package_name} that trains an
anomaly-detection model with real datasets.

— BEGIN CODE —
{code}
— END CODE —

TASK:
1. Replace **all data-loading operations** (DataLoader, torch.load, np.load,
pandas.read*, etc.)
with code that creates SMALL synthetic data directly in the script:
  • For PyOD:...
  • For PyGOD:...
  • For tslib:...
2. Keep the variable names and the rest of the logic unchanged.
3. Output runnable Python **code only** (no explanations, no markdown).
```

Table 8: This prompt directs the **Reviewer** agent to transform a full training script into a self-contained unit test. It instructs the LLM to replace all external data-loading operations with specific, library-aware code snippets that generate small synthetic datasets on the fly.

```
=== [Optimizer] ReAct prompt ===
You are an expert Python engineer specialising in anomaly-detection libraries.

Current implementation
--------
{code}

Current parameters
------
{parameter}

Current output
-----
{std_output}

Authoritative documentation
---------
{algorithm_doc}

You have access to a single tool:
“‘execute_code(params: Dict[str, Any]) -> str’” which runs the script with the
supplied **new** parameters and returns the console output.

Follow the **ReAct** loop **STRICTLY** - each response must be Either:

1. A pair of lines:
Thought: <reasoning>
Action: ‘execute_code({‘param’: value, ...})’

2. A single line starting with ‘Final:’ when you determined the final answer.

IMPORTANT:
1. Do not input ‘default’ in the parameters, use the default values from the
code.
```

Table 9: Raw ReAct (Yao et al., 2023) prompt used by the **Optimizer** agent. The prompt guides parameter tuning via strict reasoning-action loops. All content is passed to the LLM exactly as shown.