LeJOT-AutoML: LLM-Driven Feature Engineering for Job Execution Time Prediction in Databricks Cost Optimization

Lizhi Ma[†‡], Yi-Xiang Hu[†], Yihui Ren[‡], Feng Wu[*†], Xiang-Yang Li[†]
[†] *University of Science and Technology of China*, Hefei, China
[‡] *Lenovo*, Beijing, China
{malizhi,yixianghu}@mail.ustc.edu.cn, renyh10@lenovo.com, {wufeng02,xiangyangli}@ustc.edu.cn

***Abstract*—Databricks job orchestration systems (e.g., LeJOT) reduce cloud costs by selecting low-priced compute configurations while meeting latency and dependency constraints. Accurate execution-time prediction under heterogeneous instance types and non-stationary runtime conditions is therefore critical. Existing pipelines rely on static, manually engineered features that under-capture runtime effects (e.g., partition pruning, data skew, and shuffle amplification), and predictive signals are scattered across logs, metadata, and job scripts—lengthening update cycles and increasing engineering overhead. We present LeJOT-AutoML, an agent-driven AutoML framework that embeds large language model agents throughout the ML lifecycle. LeJOT-AutoML combines retrieval-augmented generation over a domain knowledge base with a Model Context Protocol toolchain (log parsers, metadata queries, and a read-only SQL sandbox) to analyze job artifacts, synthesize and validate feature-extraction code via safety gates, and train/select predictors. This design materializes runtime-derived features that are difficult to obtain through static analysis alone. On enterprise Databricks workloads, LeJOT-AutoML generates over 200 features and reduces the feature-engineering and evaluation loop from weeks to 20–30 minutes, while maintaining competitive prediction accuracy. Integrated into the LeJOT pipeline, it enables automated continuous model updates and achieves 19.01% cost savings through improved orchestration.**

***Index Terms*—automated machine learning, large language model, multi-agent systems, job orchestration, cost optimization.**

I. INTRODUCTION

Large language models (LLMs) and agentic reasoning frameworks (e.g., ReAct [1]) have advanced automated code generation and document understanding [2]–[4]. However, enterprise machine-learning (ML) workflows for regression and forecasting still rely on manual feature engineering and brittle glue code, which slows adaptation to workload drift and platform evolution [5]. This challenge is pronounced in Databricks job orchestration, where a small prediction error can translate into repeated mis-provisioning across thousands of daily runs.

LeJOT [6] is a Databricks-based framework that minimizes execution cost under dependency and latency constraints by selecting resource configurations using an execution-time predictor [7], [8]. In production, the predictor must generalize across heterogeneous instance types, changing software stacks, and non-stationary data characteristics. Four obstacles arise in this setting. First, high-impact performance signals only emerge at runtime [9]: scan volume after partition pruning, skew-induced stragglers, shuffle amplification, and executor scheduling effects. Second, these signals are fragmented across multiple sources (logs, metadata, job scripts, and configuration histories), which complicates the end-to-end feature pipeline. Third, manual feature engineering demands domain expertise in Spark SQL and platform internals, and the resulting features often lag behind evolving workloads. Finally, slow retraining and validation cycles yield stale predictors under drift [10], which degrades orchestration quality and cost efficiency.

To address these limitations, we propose LeJOT-AutoML, an agent-driven ML pipeline that turns the conventional lifecycle [11], [12] into a dynamic, self-improving system [13], [14]. A Feature Analyzer Agent (FAA) retrieves domain knowledge via retrieval-augmented generation (RAG) [15] and proposes candidate feature templates that map to observable artifacts. A Feature Extraction Agent (FExA) then invokes a Model Context Protocol (MCP) toolchain [16], [17] (log parsers, metadata queries, and a read-only SQL sandbox) to materialize both static and runtime-derived features with execution-time validation. A feedback-enabled Feature Evaluation Agent (FEvA) evaluates feature quality and model performance, and iteratively refines the pipeline to accelerate adaptation under drift [10], [18]–[20]. Two safety gates—a code-completion checker and a data-leakage checker—filter invalid extractors and prevent label leakage.

Our contributions are summarized as follows:

- **LLM-powered AutoML pipeline for enterprise job runtime prediction.** We embed LLM agents across analysis, tool invocation, feature extraction, validation, training, and model selection to enable rapid retraining and inference-time feature materialization.
- **Agent–tool collaborative feature extraction via MCP.** By combining LLM planning with tool-based execution and verification, LeJOT-AutoML efficiently extracts dynamic features that are inaccessible to purely static analysis.
- **Iterative evaluation loop with safety gates.** We introduce a feedback-driven evaluation agent and safety checks (code-completion and data-leakage detection) to

*Corresponding author.

improve reliability and drive iterative refinement until predefined criteria [18]–[20] are met.

II. Background and Motivation

LeJOT [6] performs cost-aware orchestration on Databricks by recommending resource configurations that minimize execution cost while satisfying dependency and latency constraints. The orchestration relies on two coupled components: (i) execution-time estimation under candidate resource allocations and (ii) an optimizer that selects the lowest-cost configuration that meets predicted-time constraints. Prediction errors therefore propagate directly into orchestration: underestimation violates latency Service Level Objectives (SLOs), while overestimation drives over-provisioning and recurring cost waste.

A central obstacle is that the effective processed data volume of SQL workloads is largely determined at runtime. Although data volume is a strong predictor of execution time, it does not align with static metadata (e.g., table row counts). Actual scan and shuffle volumes depend on query logic, data distributions, and optimizer behavior, which invalidates many static proxies. Static feature engineering thus fails in the following scenarios. 1) Partition pruning: Queries over partitioned tables scan a subset of data; using the total table size overestimates scan volume by orders of magnitude. 2) Join skew: Skewed key distributions concentrate work on a subset of tasks, producing stragglers and violating linear scaling assumptions. 3) Aggregation-induced shuffle: Operators like `GROUP BY` trigger intermediate shuffles; input table size underestimates network and compute costs under shuffle amplification.

These cases show that "hidden" runtime features—selectivity, skew severity, shuffle degree, and stage-level variance—are critical for accuracy yet hard to capture with fixed, hand-written extraction logic. Moreover, the relevant evidence is distributed: the SQL text encodes logical operators, the execution plan and logs encode physical behavior, and the metadata store encodes schema and partition structure. A practical pipeline must bridge these sources and update quickly when workloads drift [10]. LeJOT-AutoML addresses this gap by using LLM agents to synthesize and revise feature extractors grounded in tool-executed evidence (log parsing, metadata queries, and sandboxed SQL analysis), then validating them with safety gates before training.

III. LeJOT-AutoML Framework

A. Overview

Figure 1 shows the LeJOT-AutoML architecture and its integration into the LeJOT pipeline. The framework operates in two phases: (i) an automated training phase that generates and validates artifacts, and (ii) an online inference phase for real-time prediction.

Training Phase. LeJOT-AutoML forms a closed-loop AutoML pipeline consisting of five core components: the Feature Analyzer Agent (FAA), Feature Extraction Agent (FExA), a baseline model, the Feature Evaluation Agent (FEvA), and a model selector. Two safety gates—a *code-completion checker* and a *data-leakage checker*—ensure that generated feature-extraction code is executable and does not leak label information.

The training workflow starts with FAA, which ingests three heterogeneous sources: execution logs (e.g., shuffle volumes and stage/task runtimes), a domain knowledge base (e.g., Spark SQL practices), and a metadata store (e.g., schema, partitions, and data statistics). Guided by retrieval-augmented generation (RAG) [21], FAA proposes a structured feature list. FExA then generates extraction programs and materializes feature values through an MCP toolchain (e.g., metadata queries and a read-only SQL sandbox). After passing safety checks, the resulting feature matrix is used to train baseline and candidate models. FEvA evaluates feature quality (coverage, skewness), feature utility (importance and redundancy), and model-level metrics, then emits actionable feedback for refinement. Finally, the model selector chooses the best-performing model (e.g., XGBoost [3], LightGBM [22]) and hyperparameters for deployment.

Inference Phase. For a new job, FAA reuses learned feature templates to determine the required feature set and the corresponding extraction plan. FExA extracts features in parallel from scripts/metadata and via lightweight sandbox analysis. The standardized feature vector is fed into the deployed predictor to estimate execution time, which is then consumed by LeJOT's orchestration algorithm to select a cost-minimizing configuration. Prediction residuals and feature health signals are logged and fed back into the training loop for continuous updates.

B. System interfaces and design goals

LeJOT-AutoML sits between raw job artifacts and the downstream orchestration algorithm. Its inputs consist of (i) static artifacts (job code, configuration, cluster specification, metadata), (ii) runtime traces (logs and metrics), and (iii) an evolving knowledge base that stores domain rules and "feature experience" derived from previous runs. The outputs consist of a versioned feature specification and a deployed predictor with a reproducible extraction bundle.

We follow four design goals. **(1) Low-latency inference:** extraction plans prioritize features with bounded runtime cost, and the toolchain executes reads under a strict sandbox policy. **(2) Safety and governance:** every generated extractor passes syntactic completeness checks and leakage screening before execution. **(3) Continuous adaptation:** model retraining is triggered by drift signals or a periodic schedule, reducing staleness in dynamic workloads. **(4) Traceability:** each feature records provenance (source, transformation, and collection method), which supports debugging and compliance.

C. Mathematical Formulation

Let $\mathcal{D}_t = \{d_i\}_{i=1}^{n_t}$ denote the dataset at time t, where each instance $d_i = (d_i^s, d_i^u)$ contains structured and unstructured information, and let $\mathbf{Y}_t = \{y_i\}_{i=1}^{n_t}$ be the observed execution times. Given a knowledge base $\mathcal{K}$ and an MCP toolset $\mathcal{T}$, the

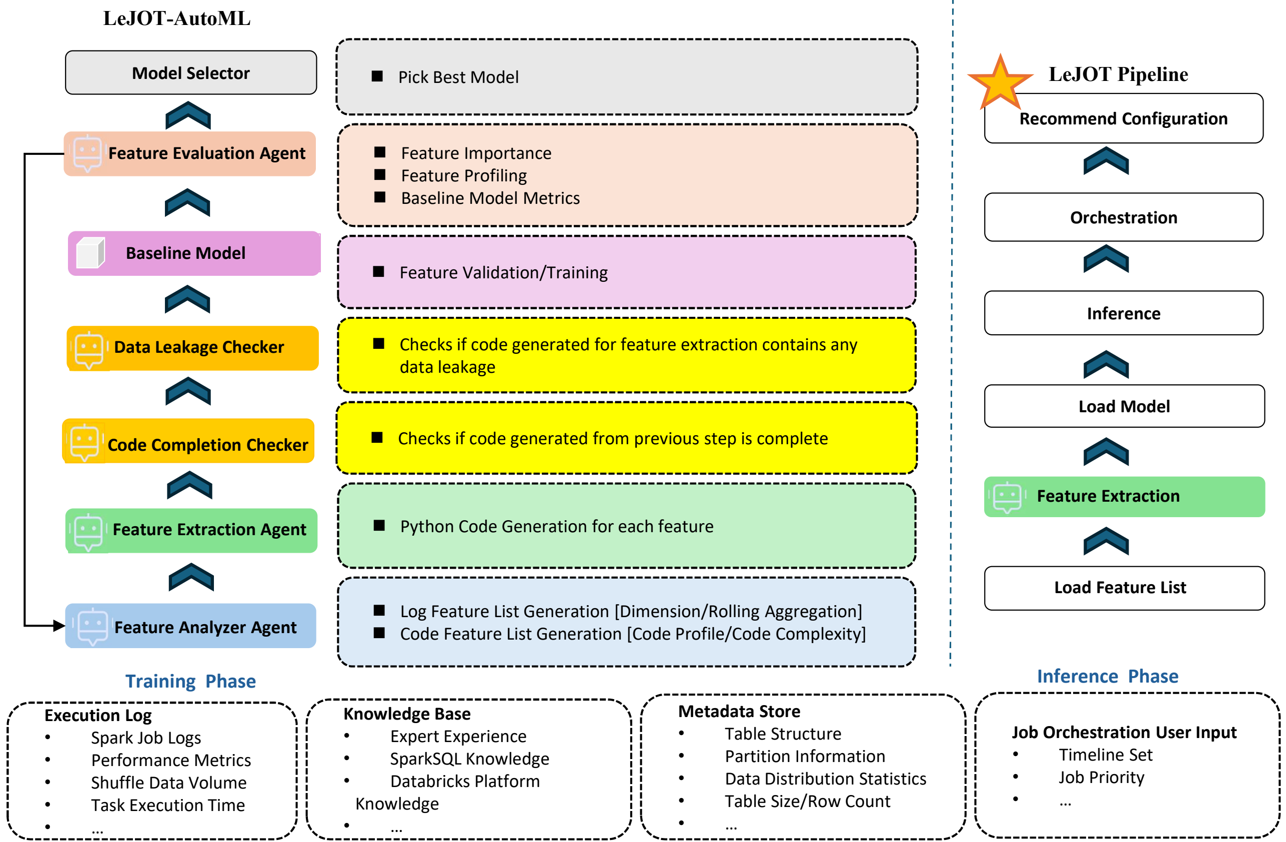

Fig. 1. Overview of LeJOT-AutoML. (a) Left: the end-to-end LeJOT-AutoML system. (b) Right: how LeJOT-AutoML integrates into the LeJOT pipeline.

agent performs feature analysis ϕ_{analyze} and feature extraction ϕ_{extract}. For each instance, the agent retrieves the domain context via RAG:

$$R_i = \text{RAG}(Q(d_i), \mathcal{K}), \tag{1}$$

and determines a job-specific feature set:

$$F_i = \phi_{\text{analyze}}(d_i, R_i). \tag{2}$$

Feature values are materialized via tool invocation:

$$\mathbf{x}_i = \{\phi_{\text{extract}}(f, d_i, \mathcal{T}) \mid f \in F_i\}. \tag{3}$$

Stacking all instances yields the feature matrix $\mathbf{X}_t = [\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_{n_t}]^\top$. A predictor $M(\cdot;\theta)$ is trained by minimizing a loss $\mathcal{L}$:

$$\theta_t^* = \arg\min_{\theta} \mathcal{L}(\mathbf{X}_t, \mathbf{Y}_t; \theta), \tag{4}$$

and the deployed model $M_t(\cdot;\theta_t^*)$ predicts $\hat{y} = M_t(\mathbf{x}_{\text{new}})$ for a new job d_{new}.

Cost-aware feature selection. Online extraction introduces a latency budget that constrains the usable feature set. Let $c(f, d)$ denote the extraction cost of feature f on job artifact d, and let B be a per-request budget. FAA therefore targets feature sets that jointly improve prediction accuracy and satisfy runtime constraints:

$$\begin{aligned} F_{\text{new}} = \arg\min_{F \subseteq \mathcal{F}} \; & \mathbb{E}[\ell(M(\mathbf{x}_F), y)] + \lambda \sum_{f \in F} c(f, d_{\text{new}}) \\ \text{s.t.} \; & \sum_{f \in F} c(f, d_{\text{new}}) \le B, \end{aligned} \tag{5}$$

where $\mathcal{F}$ is the global feature universe, $\mathbf{x}_F$ denotes the subvector restricted to F, and λ controls the accuracy–latency tradeoff.

Safety constraints. Each generated extractor program p_f is executed only if it passes two gates: a syntactic completeness predicate $g_{\text{cc}}(p_f) = 1$ and a leakage predicate $g_{\text{dl}}(p_f) = 1$. These gates impose hard constraints on feasible feature sets:

$$\forall f \in F: \quad g_{\text{cc}}(p_f) = 1 \;\wedge\; g_{\text{dl}}(p_f) = 1. \tag{6}$$

Continuous updates. When new data $\mathcal{D}_{\text{new}}$ arrives, the system updates $\mathcal{D}_{t+1} = \mathcal{D}_t \cup \mathcal{D}_{\text{new}}$, $\mathbf{Y}_{t+1} = \mathbf{Y}_t \cup \mathbf{Y}_{\text{new}}$, and retrains:

$$\theta_{t+1}^* = \arg\min_{\theta} \mathcal{L}(\mathbf{X}_{t+1}, \mathbf{Y}_{t+1}; \theta). \tag{7}$$

This cycle is triggered periodically (e.g., daily) or by drift signals to maintain reliable predictions under workload evolution.

D. Implementation Details

We summarize implementation choices that make LeJOT-AutoML practical in an enterprise setting, focusing on (i) the MCP tool interface, (ii) execution and caching policies, and (iii) feedback and drift handling.

a) MCP tool interface and outputs: Each MCP tool exposes a restricted, typed interface and returns structured outputs (e.g., JSON-like records) that can be deterministically transformed into features. We group tools into three categories: (i) metadata tools (schema, partition layout, table statistics, cluster configuration), (ii) log/trace tools (stage/task timing, shuffle read/write, spill metrics, failure reasons), and (iii) sandbox tools that execute read-only SQL queries or lightweight plan inspection under strict policies. For robustness, each extractor emits a typed feature schema (name, type, default value, provenance), and the Feature Extraction Agent validates tool outputs against the schema before materialization.

b) Execution policy (safety, determinism, parallelism): All generated extractors run in a sandboxed environment with an allowlist of libraries and tool calls. The code-completion checker blocks extractors with missing imports, undefined variables, or unresolved tool outputs. The data-leakage checker enforces availability: a feature must be computable from information available before a scheduling decision (e.g., job scripts and historical traces), and cannot rely on post-run artifacts. For efficiency, independent tool calls are executed in parallel when dependencies permit. We also bound inference latency by capping per-job sandbox queries and enforcing timeouts.

c) Caching and versioning: To avoid repeated reads for recurring jobs, we cache intermediate tool outputs and materialized feature values. Cache keys are tuples of (job signature, data snapshot identifier, feature version, tool version), enabling safe reuse and incremental retraining by re-materializing only affected features. Each deployed model is packaged with a versioned feature specification and extractor bundle to ensure online inference replays training-time transformations.

d) Feedback signals and drift-triggered updates: LeJOT-AutoML logs prediction residuals, feature health signals (missingness, outliers, schema mismatches), and extraction latency per tool modality. These signals drive periodic refresh (e.g., daily retraining) and drift-triggered refresh when residuals or feature distributions exceed thresholds. During retraining, FEvA summarizes failure modes (unstable features, high-cardinality categoricals, redundancy) and feeds concise guidance to FAA/FExA to refine feature specifications and extraction plans.

IV. Design of Core Module Functions

This section details the core modules that implement the closed-loop lifecycle in Section III.

A. Feature Analyzer Agent (FAA)

The FAA determines the candidate feature space and therefore the accuracy ceiling of the predictor. It takes as input: (i) the task objective (execution-time prediction for cost-aware orchestration), (ii) supplementary artifacts (job scripts, configuration, historical logs, cluster and table metadata), (iii) constraints (collection cost, privacy policy, and access scope), and (iv) an output schema. Using this information, FAA performs two functions.

(1) Context recovery via RAG. FAA formulates queries over $\mathcal{K}$ to retrieve Spark SQL and platform knowledge that clarifies which runtime behaviors dominate performance. Retrieved context is then grounded against job artifacts to avoid generic suggestions.

(2) Feature specification synthesis. FAA outputs a list of *feature specifications* rather than raw names. Each specification records: *name*, *type* (numerical/categorical/text-derived), *source* (log/metadata/code), *extraction plan* (tool calls and transformations), and *expected cost* and *refresh frequency*. This schema supports traceability, caching, and downstream parallel extraction.

B. Feature Extraction Agent (FExA)

The FExA materializes the proposed features via an agent–tool collaborative architecture. The LLM performs planning and program synthesis, while the MCP toolchain executes and verifies retrieval steps through a constrained interface [16]. FExA follows a four-stage extraction pipeline:

- **Static extraction:** parses job scripts and metadata to obtain invariant features (e.g., operator counts, join patterns, table statistics, partition layouts).
- **Runtime materialization:** invokes log parsers and the read-only SQL sandbox to collect runtime-derived signals (e.g., stage imbalance, shuffle amplification, pruning effectiveness).
- **Normalization and encoding:** maps heterogeneous outputs into a unified feature vector through scaling, one-hot encoding, and text vectorization when needed.
- **Data-quality checks:** flags missing values, outliers, and schema mismatches, then emits repair actions (fallback features, default values, or re-execution with tightened queries).

To meet inference latency goals, FExA executes independent extractors in parallel and uses caching keyed by (job signature, feature version, and data snapshot), which reduces repeated reads across similar recurring jobs.

C. Feature Evaluation Agent (FEvA)

The FEvA performs a multi-level assessment to decide which features enter the deployed model. It aggregates three families of signals:

Feature health. For each feature, FEvA measures coverage (missing rate), stability (variance under similar jobs), and distribution shifts across time windows, identifying brittle or non-stationary features.

Feature utility. FEvA estimates importance and redundancy using model-based attribution (gain/SHAP-style summaries from tree models) and correlation screening, which reduces collinearity and overfitting risks.

TABLE I
FEATURE DIVERSITY COMPARISON

	AutoML	Manual
Number of Features	200+	40+
Feature Types	- Log Profiling Features - Historical Time Series Data - Driver Node Historical Data	- Node Configuration Historical Data - Log Profiling Features

TABLE II
TOP-5 FEATURE IMPORTANCE FOR AUTOML VS. MANUAL

Feature Name	AutoML (%)	Manual (%)
duration_seconds_lag_1	28.8	–
vcpu_lag_1	24.6	–
duration_seconds_shifted_avg_last_3_runs	8.0	–
DBU_lag_1	7.8	–
Memory_lag_1	3.6	–
vcpu_ratio	–	16.2
vmemory_ratio	–	13.4
total_memory_changed	–	8.0
total_cpu_changed	–	7.9
worker_flexibility_ratio	–	4.1

End-to-end impact. FEvA evaluates the marginal impact of candidate features through ablations on baseline models and reports deltas in MAE/RMSE and R^2. The output is a structured feedback packet that instructs FAA/FExA to refine extraction plans, drop unstable features, or propose additional domain-grounded interactions.

D. Safety gates and model selection

LeJOT-AutoML executes generated code under strict safety gates. The **code-completion checker** verifies that each extractor program is syntactically complete, imports only approved libraries, and returns values conforming to the feature schema. The **data-leakage checker** enforces temporal and semantic isolation between features and labels, rejecting extractors that directly access the target (execution time) or indirectly derive it from post-run artifacts.

After FEvA produces evaluation summaries, the **model selector** searches a bounded candidate set of algorithms and hyperparameters, trains candidates on the versioned feature matrix, and selects the final configuration for deployment. The selected model is packaged together with its feature specification and extractor bundle, ensuring that online inference replays the same transformations used during training.

V. EXPERIMENTS

A. Computing Environment

Experiments were conducted on a single machine with an Intel Core Ultra 7 165U CPU, 32 GB RAM, and Windows 11 Professional. We used Qwen3-235B-A22B ("Qwen-235B") [23] for agent reasoning and feature/code synthesis, and trained the execution-time predictor using XGBoost [3].

B. Methodology

We evaluate LeJOT-AutoML on enterprise Databricks workloads by comparing *AutoML* (LLM+MCP automated feature engineering) with *Manual* feature engineering. The pipeline follows an analysis–act–validate loop: the LLM parses unstructured job artifacts (scripts and logs), the MCP toolchain materializes runtime-derived signals, and safety checks validate extracted features. We use 5-fold cross-validation to estimate generalization and conduct ablation studies to quantify the contributions of different feature sources. We report feature diversity, prediction metrics (MAE, MAPE, R^2), per-module runtime, and end-to-end cost savings in LeJOT.

C. Manual vs. AutoML

1) Feature Analysis: AutoML generates 200+ features spanning log profiling, time-series statistics, and driver-node history, whereas manual engineering produces roughly 40 features primarily derived from node-configuration history (Table I). The top-ranked features differ substantially between the two approaches (Table II), suggesting that AutoML captures additional temporal and workload-dependent signals beyond static resource ratios. Using the same training and test datasets, Table I compares feature diversity. AutoML generates over 200 features, spanning log profiling, time-series signals, and Driver-node historical signals. In contrast, Manual ML produces around 40 features, primarily focused on node configuration history.

AutoML completes three iterations within 20–30 minutes, compared with roughly one month for manual engineering (Table III). While manual features achieve higher accuracy (R^2=0.91 vs. 0.81), AutoML provides competitive performance at a small fraction of the development cost. A representative case study (Table IV) shows that manual predictions decrease sharply when upgrading instance types and enabling Photon, while AutoML predictions remain relatively stable, indicating that AutoML currently under-models the direct impact of resource upgrades.

D. Additional results

Module-level runtime (Table V) shows that FAA and the data-leakage checker dominate end-to-end latency, reflecting the cost of LLM reasoning and semantic verification. Over three FEvA iterations, MAE decreases from 247.95 to 145.64 and R^2 improves from 0.61 to 0.81 (Table VI), validating the

TABLE III
COMPARISON OF AUTOML AND MANUAL FEATURE EXTRACTION APPROACHES

Metrics	AutoML	Manual
R^2	0.81	0.91
MAPE	20.13%	19.49%
MAE	123.29	78.94
Time	20−30 min for 3 Iterations	1 Month

TABLE IV
INFERENCE RESULTS FOR A REPRESENTATIVE JOB UNDER TWO FEATURE ENGINEERING METHODS

Compute Machine	AutoML (s)	Manual (s)
Standard_F4s	167	206
Standard_F16s	154	82
Standard_E16_v4	162	78
Standard_F16_v4_Photon	147	52

TABLE V
EXECUTION TIME (IN SECONDS) FOR EACH AGENT NODE ACROSS FIVE EXPERIMENTAL RUNS.

Agent	Run 1	Run 2	Run 3	Run 4	Run 5
Feature Analyzer	260.60	233.27	263.29	223.47	298.12
Feature Extractor	107.80	92.71	105.25	115.79	137.87
Code Completion Checker	<0.01	<0.01	<0.01	<0.01	<0.01
Data Leakage Checker	199.89	172.21	209.45	210.34	325.6
Evaluation Agent	33.39	38.85	43.62	49.66	38.80
Model Selector	27.12	20.87	34.56	24.93	32.43

TABLE VI
METRICS OF THE BASELINE XGBOOST MODEL OVER THREE ITERATIONS OF EVALUATION

Metric	First Iteration	Second Iteration	Third Iteration
MAE	247.95	172.09	145.64
MAPE (%)	36.28	25.18	21.31
R^2	0.61	0.80	0.81

effectiveness of the feedback loop. Integrated into LeJOT, AutoML achieves 19.01% cost savings (Table VII), demonstrating practical value even with a modest accuracy gap.

VI. CONCLUSION AND DISCUSSION

We presented LeJOT-AutoML, an LLM-driven framework for automated feature engineering in Databricks job execution-time prediction. By integrating LLM agents with an MCP toolchain, the system expands the feature space to include hard-to-observe runtime signals and compresses the feature-engineering cycle from months to minutes. Although manual feature engineering still delivers better generalization across hardware configurations (R^2=0.91 vs. 0.81), LeJOT-AutoML provides a scalable, low-maintenance alternative that enables continuous learning and achieves 19.01% cost savings in LeJOT. Future work will focus on improving resource awareness by incorporating richer configuration- and runtime-level indicators of execution and data-movement behavior.

ACKNOWLEDGMENTS

The research is partially supported by Innovation Program for Quantum Science and Technology 2021ZD0302900 and China National Natural Science Foundation with No.62132018, 62231015, "Pioneer" and "Leading Goose" R&D Program of Zhejiang, 2023C01029, and 2023C01143, Anhui Provincial Natural Science Foundation under Grant 2208085MF172 and the USTC Kunpeng-Ascend Scientific and Educational Innovation Excellence Center. We also thank the anonymous reviewers for their comments and helpful feedback.

TABLE VII
COST SAVING RATE COMPARISON BETWEEN SOLUTIONS

Solution	Throughput (k/s)	Initial Cost ($k)	Final Cost ($k)	Cost Saving Rate
AutoML	0.08	52.6	42.6	19.01%
Manual ML	0.12	52.6	37.9	27.94%

REFERENCES

[1] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao, "React: Synergizing reasoning and acting in language models," *arXiv preprint arXiv:2210.03629*, 2022.

[2] B. Wang, X. Du, Y. Bai *et al.*, "A survey on large language models as agents," *arXiv preprint arXiv:2309.07864*, 2023.

[3] T. Chen and C. Guestrin, "XGBoost: A scalable tree boosting system," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, 2016.

[4] J. Li, Q. Liu, K. Zhang, and M. Wang, "Agent-based automated machine learning for enterprise applications," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 34, no. 8, pp. 4567–4580, 2023.

[5] M. Feurer, K. Eggensperger, S. Falkner, and F. Hutter, "Auto-sklearn 2.0: Hands-free automl via meta-learning," *Journal of Machine Learning Research*, vol. 23, no. 261, pp. 1–61, 2022. [Online]. Available: https://jmlr.org/papers/v23/21-1100.html

[6] L. Ma, Y.-X. Hu, Y. Wang, Y. Zhao, Y. Ren, J.-X. Liao, F. Wu, and X.-Y. Li, "Lejot: An intelligent job cost orchestration solution for databricks platform," in *2025 11th International Conference on Big Data Computing and Communications (BigCom)*, 2025.

[7] A. Kipf, T. Kipf, B. Radke *et al.*, "Learned cardinalities: Estimating correlated joins with deep learning," in *Conference on Innovative Data Systems Research (CIDR)*, 2019.

[8] J. Park, N. Polyzotis, S. Roy *et al.*, "Learning-based query cardinality estimation with deep neural networks (naru)," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020. [Online]. Available: https://dl.acm.org/doi/10.1145/3318464.3389711

[9] S. Kim, J. Park, M. Lee, and Y. Choi, "Dynamic feature extraction for real-time machine learning inference," *IEEE Transactions on Knowledge and Data Engineering*, vol. 35, no. 12, pp. 3456–3468, 2023.

[10] S. Venkataraman, Z. Yang, D. Liu *et al.*, "Ernest: efficient performance prediction for large-scale advanced analytics," *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pp. 363–378, 2016.

[11] E. LeDell and S. Poirier, "H2o automl: Scalable automatic machine learning," *arXiv preprint arXiv:2004.11731*, 2020.

[12] Y. Peng, H. Raghavan, S. Venkataraman *et al.*, "Slaq: Quality-driven scheduling for distributed machine learning training," in *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, 2018.

[13] X. He, K. Zhao, and X. Chu, "Automated machine learning: methods, systems, challenges," *Automated Machine Learning: Methods, Systems, Challenges*, pp. 3–19, 2019.

[14] M. Feurer, A. Klein, K. Eggensperger, J. Springenberg, M. Blum, and F. Hutter, "Auto-sklearn: efficient and robust automated machine learning," *Automated Machine Learning: Methods, Systems, Challenges*, pp. 113–134, 2019.

[15] M. Liu, S. Chen, P. Wang, and L. Zhang, "Rag-enhanced feature engineering for machine learning pipelines," *Proceedings of the ACM on Management of Data*, vol. 2, no. 1, pp. 1–25, 2024.

[16] Anthropic, "Model context protocol (mcp) specification," https://modelcontextprotocol.io, 2024, accessed 2025-10-09.

[17] T. Schick, J. Dwivedi-Yu, R. Raileanu *et al.*, "Toolformer: Language models can teach themselves to use tools," *arXiv preprint arXiv:2302.04761*, 2023.

[18] L. Zhang, M. Wang, H. Chen, and W. Liu, "Job execution time prediction in distributed computing systems: a survey," *ACM Computing Surveys*, vol. 56, no. 3, pp. 1–35, 2023.

[19] K. Wang, M. M. H. Khan, N. Nguyen, and S. Gokhale, "Spark performance prediction using machine learning," *Cluster Computing*, vol. 24, no. 3, pp. 1921–1935, 2021.

[20] J. Li, M. Bäckström, B. van Stein, A. Biedenkapp, F. Hutter, and M. Lindauer, "Large language models for automated data science: introducing caafe for context-aware automated feature engineering," *arXiv preprint arXiv:2309.03428*, 2023.

[21] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel *et al.*, "Retrieval-augmented generation for knowledge-intensive nlp tasks," *Advances in Neural Information Processing Systems*, vol. 33, pp. 9459–9474, 2020.

[22] G. Ke, Q. Meng, T. Finley *et al.*, "Lightgbm: A highly efficient gradient boosting decision tree," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

[23] A. Yang *et al.*, "Qwen3 technical report," *arXiv preprint arXiv:2505.09388*, 2025.