RESEARCH ARTICLE

Scalable Software Engineering Architecture for AI-Enabled ETL Pipelines Using Event-Driven Microservices

Karthik Babu Manam[1,*]

[1] University of the Cumberlands, Williamsburg 40769, United States

Abstract

As data ecosystems become more diverse and time-critical, traditional monolithic ETL pipelines face challenges to meet the demands of modern data engineering workloads in terms of scalability, adaptability, and operational resilience. In this paper, we introduce an event-driven microservices approach to orchestrate and deploy AI-based ETL (ETL = Extraction, Transformation, and Loading) pipelines in a Kubernetes-managed environment that includes the following components: asynchronous orchestration using Apache Kafka, hybrid anomaly detection, adaptive schema inference, and predictive load balancing. The proposed architecture breaks the ETL processing into loosely coupled services, which can be deployed and scaled independently, and incorporates data quality intelligence into the transformation layer. Two explicit baselines are used for experimental evaluation: (1) a traditional monolithic batch-processing ETL pipeline with sequential execution of the stages performed without horizontal scaling, and (2) an event-driven microservices pipeline, which uses Apache Kafka orchestration but no AI-enabled optimization modules. The framework is able to achieve 39.7% improvement in throughput (4,820 vs. 3,450 records/sec) and 96.3% anomaly detection accuracy, while reducing average E2E latency by 14.6% (245 vs. 287 ms). The framework's throughput is 288.7% higher than the baseline of ETL monolithic, and the average latency drops 72.5% to 245 ms compared to 892 ms for the baseline. An architectural assessment also shows that the system is more modular, loosely coupled, more fault isolated, and more flexible to deploy; all of which are important software quality attributes. The results indicate the proposed architecture as a potentially reusable reference to create scalable and intelligent ETL systems in enterprise data processing environments and emphasize the necessity of generalization in multi-domain validation.

Keywords: Event-Driven architecture, microservices engineering, scalable ETL pipelines, AI-Enabled data transformation, distributed data processing, predictive load balancing, anomaly detection, container orchestration, software engineering frameworks, data quality management.

Submitted: 08 May 2026
Revised: 24 June 2026
Accepted: 29 June 2026
Published: 14 July 2026

Vol. 2, **No.** 3, 2026.
10.62762/JSE.2026.382038

***Corresponding author:**
✉ Karthik Babu Manam
kmanam65453@ucumberlands.edu

Citation
Manam, K. B. (2026). Scalable Software Engineering Architecture for AI-Enabled ETL Pipelines Using Event-Driven Microservices. *ICCK Journal of Software Engineering*, 2(3), 169–184.

1 Introduction

The world of enterprise data systems is changing at an accelerated pace. Organizations have a need to scale efficiently, meet real-time analytical needs, and manage multiple and disparate data sources without undue complexity. These environments are best suited to traditional Extract, Transform, Load (ETL) systems, which are usually developed as monolithic batch-processing pipelines, are too tightly coupled, hard to horizontally scale, and slow to adapt to fluctuations in workload and schemas [1]. Since then, microservices architecture has become a popular software engineering paradigm that overcomes many of the drawbacks of monolithic applications by enabling distributed, modular, and independently deployable systems [2]. Microservices break up a large system into smaller services with well-defined interfaces, allowing each stage of the pipeline to be developed, deployed, tested, and scaled independently. In data-engineering terms, this separation provides enhanced fault isolation, technology diversity and resource-aware scaling of the expensive compute phases. Event-driven architectures work hand-in-hand with microservices, and are used to support the asynchronous and decoupled communication between microservices [3]. Temporal decoupling, buffering, replayability and natural handling of backpressure are important properties required by high throughput ETL pipelines that are provided by the distributed event brokers and streaming platforms. This pairing is especially well suited for enterprise-level data workflows where the ingestion rate is variable and extensibility and resilience are key. Meanwhile, AI and machine learning have brought in new capabilities for data processing, such as anomaly detection, adaptive schema management and workload-aware resource optimization. The AI-based elements can enhance the reliability of ETL by detecting issues with data quality as they happen, making adjustments to input streams as they change, and predicting capacity needs before performance dips. The challenge of incorporating this intelligence into distributed ETL systems involves software engineering decisions on service boundaries, orchestration patterns, and computational placement [3]. While previous research has made significant contributions in the area of microservices processing, processing with event-driven communication, anomaly detection and predictive autoscaling, these separate functions are usually not combined in a single software engineering paradigm for ETL pipelines. The value of intelligent processing built into pipeline processes is not widely stressed in most of the work available, which focuses on scalability of the infrastructure [6]. Similarly, the anomaly detection problem in distributed systems is commonly discussed in the context of operational monitoring, not data quality at the transformation layer, and the typical implementation of predictive autoscaling is at infrastructure level and without regard to pipeline semantics [7]. These limitations motivate the need for an integrated architecture that combines scalable service decomposition, event-driven orchestration, AI-assisted transformation, and predictive resource management.

Accordingly, this study is guided by the following formally stated research question and hypothesis:

Research Question (RQ): Can an event-driven microservices architecture that embeds hybrid anomaly detection, adaptive schema inference, and predictive load balancing improve ETL throughput, latency, and operational quality attributes over conventional monolithic and non-AI microservices baselines?

Hypothesis (H1): Integrating hybrid anomaly detection, adaptive schema inference, and predictive load balancing within a modular event-driven microservices design will yield statistically significant improvements in runtime performance (throughput improvement $\geq$25% and latency reduction $\geq$10% relative to the non-AI event-driven baseline) and will improve software engineering quality attributes including maintainability, scalability, and fault isolation, as evaluated against ISO/IEC 25010 criteria. The key contributions of this research are as follows:

- A modular, fault isolated, horizontally scalable microservices architecture that breaks up ETL monoliths into loosely coupled, independent deployable services and orchestrates them using asynchronous messaging using Apache Kafka.
- Adaptive schema inference with a transformation service that integrates an AI-driven Isolation Forest and Autoencoder anomaly detection, which provides intelligent data quality management without manual effort.
- Design of Predictive Load Balancing Service (PLBS) based on Gradient-based time-series forecasting to minimize latency variations in load balancing service during load variations, reducing the reaction time for scaling service to the resources.

Table 1. Comparative analysis of reviewed Literature.

Focus Area	Methodology	Scalability	AI/ML Integration	Event-Driven	Predictive Scaling	Limitations
Microservices processing patterns [8]	Service mesh orchestration	Partial	No	No	No	No intelligent transformation or predictive scaling
Serverless data pipelines [9]	Taxonomy and classification	Yes	No	Partial	No	No concrete framework; no AI-enabled processing
Cloud ETL for digital twins [6]	Rule-based ETL framework	Yes	No	No	No	Static transformations; no anomaly detection
Data ingestion patterns [11]	Metadata-driven design	Partial	No	No	No	Ingestion only; no downstream intelligence
Stream processing autoscaling [7]	Reactive adaptive scaling	Yes	No	Partial	Reactive only	No predictive forecasting; no data quality awareness
Anomaly detection at scale [13]	ML-based monitoring	Yes	Yes	No	No	Infrastructure monitoring only; not ETL-integrated
Distributed big data architecture [14]	Domain-driven reference model	Yes	No	No	No	No event-driven communication; no AI transformation
ETL approaches comparison [15]	Comparative survey	Partial	No	No	No	Survey only; no unified framework proposed
Predictive autoscaling [16]	Continual learning forecasting	Yes	Yes	No	Yes	Infrastructure-level only; no pipeline integration
AI-enabled ETL with event-driven microservices [Proposed]	Hybrid anomaly detection + predictive load balancing	Yes	Yes	Yes	Yes	Addresses all identified gaps in unified framework

- The Online Retail II e-commerce transaction data set was used to comprehensively evaluate the experiments, with the results assessed for anomaly detection accuracy, pipeline throughput, latency, and stability across multiple runs, as well as horizontal service replication to test scalability.

Together these contributions underscore the potential for a viable, scalable, and practically deployable architecture for modern enterprise ETL systems, using event-driven microservices principles, combined with AI-powered data transformation and predictive resource management.

2 Literature Review

As data engineering needs are growing at scale, cloud-native software architectures are becoming increasingly popular, and intelligent analytics are gaining traction, a huge volume of research has been done around microservices-based processing systems, event-driven pipelines, ETL modernization, predictive autoscaling, and anomaly detection. This section summarizes some of the representative studies that were conducted in the past few years, highlighting their applicability to distributed data integration, intelligent data transformation, and scalable orchestration. The review will be representative and not systematic literature review—papers were chosen to represent important architectural/methodological strands that are most directly related to the proposed framework.

Steering the processing pattern service by integrating it into the microservices application, Garcia-Valls et al. [8] proposed an approach called service mesh for cloud-native data processing environments. They enhanced the flexibility of orchestration and decrease of coupling between processing components in containerized deployments. But the framework didn't include any aspect of intelligent ETL scenarios such as AI data transformation and predictive scaling. Wen et al. [9] proposed a taxonomy of the various serverless data pipeline approaches and provided definitions of orchestration strategies, execution models, and scalability properties in cloud-native processing. Their analysis revealed the benefits of elastic scaling and cost efficiency, but was largely classificatory without suggesting a unified architecture that combines anomaly detection or predictive workload management into ETL pipelines. The ETL framework proposed by Dapkute et al. [6] for digital twin data management was implemented on the cloud and its ETL throughput and schema mapping performances were evaluated for different amounts of digital twin data. The system did not feature automated anomaly detection, predictive load balancing functions, and only offered static transformations with real-time synchronization and mixed batch-stream processing.

To enhance the efficiency of ingesting data in cloud-based systems, Khan et al. [11] suggested a design pattern approach supported by metadata. Their framework improved the handling of schema evolution and engineering flexibility for data ingestion, but they were limited to only the extraction layer and failed to provide any intelligence in the downstream transformation, orchestration of services and dynamic scaling. Rodrigues et al. [12] conducted a systematic literature review of performance and scalability assessment in microservice architectures. Their

synthesis revealed that scalability strategies for microservice deployments are frequently evaluated through throughput and latency benchmarks, yet comparatively few studies jointly address predictive scaling and data quality semantics within the same architectural framework.

Wang et al. [13] created an anomaly detection system for incident response in an enterprise scale using machine learning approach. Their framework showed good performance in real-time anomaly detection on distributed operational metrics, but it did not apply to the ETL system transformation layer data quality monitoring. Ataei [14] made a reference architecture of distributed big data systems, which defines bounded contexts for ingestion, processing, storage, and serving of big data. The model did not cover event-driven asynch messaging, artificial intelligence-driven transformation, or predictive workload forecasting.

Bouassida et al. [15]analysed the differences between ETLs in traditional and big data architectures, and observed that the requirements for scalability, real-time processing, and schema flexibility are changing. Their study provided a helpful comparison but didn't suggest any single architectural solution or incorporate machine learning tools for data quality management. Hao et al. [16] proposed a continual learning approach to predictive autoscaling under dynamic cloud workloads. Their work addressed concept drift in forecasting models and reduced over-provisioning relative to static retraining. Nevertheless, the proposed method remained infrastructure-centric, without direct integration into event-driven microservices-based ETL workflows.

More recent work reinforces the continuing relevance of intelligent orchestration and software quality concerns in distributed data systems. Recent studies on data pipeline quality and the application of microservices patterns to large-scale data systems emphasize the importance of integrating maintainability, reliability, and data quality considerations into pipeline design and deployment [1, 2]. These works indicate a trend toward more software-engineering-aware data pipeline designs, but they still stop short of tightly coupling anomaly-aware transformation, schema adaptation, and predictive scaling within a single ETL reference architecture.

Beyond architectural capabilities, software quality models are also relevant for evaluating ETL frameworks. Table 1 summarizes the comparative analysis of the reviewed literature across these dimensions. ISO/IEC 25010 identifies maintainability, reliability, performance efficiency, and compatibility as central software product quality attributes. In the context of event-driven microservices, maintainability is reflected in service modularity and independent deployability, reliability in fault isolation and recovery behavior, and performance efficiency in throughput and latency under changing workloads. Many prior ETL studies report throughput or resource metrics, but comparatively few frame their architectural decisions in terms of such quality attributes.

The reviewed literature shows that prior research has generally treated microservices architecture, anomaly detection, and autoscaling as separate concerns. Service mesh and distributed architecture studies [8, 14] improved modular processing but omitted intelligent transformation and predictive resource management. ETL and data pipeline studies [9, 13] emphasized execution models and scalability but not transformation-layer intelligence. Anomaly detection studies such as [13] focused on infrastructure behavior rather than data quality embedded in ETL stages. Autoscaling work [12, 16] improved resource allocation but remained largely infrastructure-focused.

Importantly, previous attempts that approached parts of this integration were still insufficient because they did not jointly address three issues: first, transformation-layer anomaly handling that operates on business data rather than only operational telemetry; second, schema evolution management that preserves compatibility across independently versioned consumers; and third, predictive scaling decisions informed by pipeline behavior rather than solely infrastructure counters. The proposed study builds on these prior advances by combining these concerns into a unified event-driven microservices architecture designed specifically for AI-enabled ETL systems.

3 Materials and Methods

As illustrated in Figure 1, the proposed scalable software engineering framework for AI-enabled ETL pipelines consists of a Data Ingestion Service, AI-Enabled Transformation Service, Predictive Load Balancing Service, Load and Delivery Service, and a Container Orchestration Layer. The architecture is organized as a loosely coupled microservices system in which communication occurs exclusively

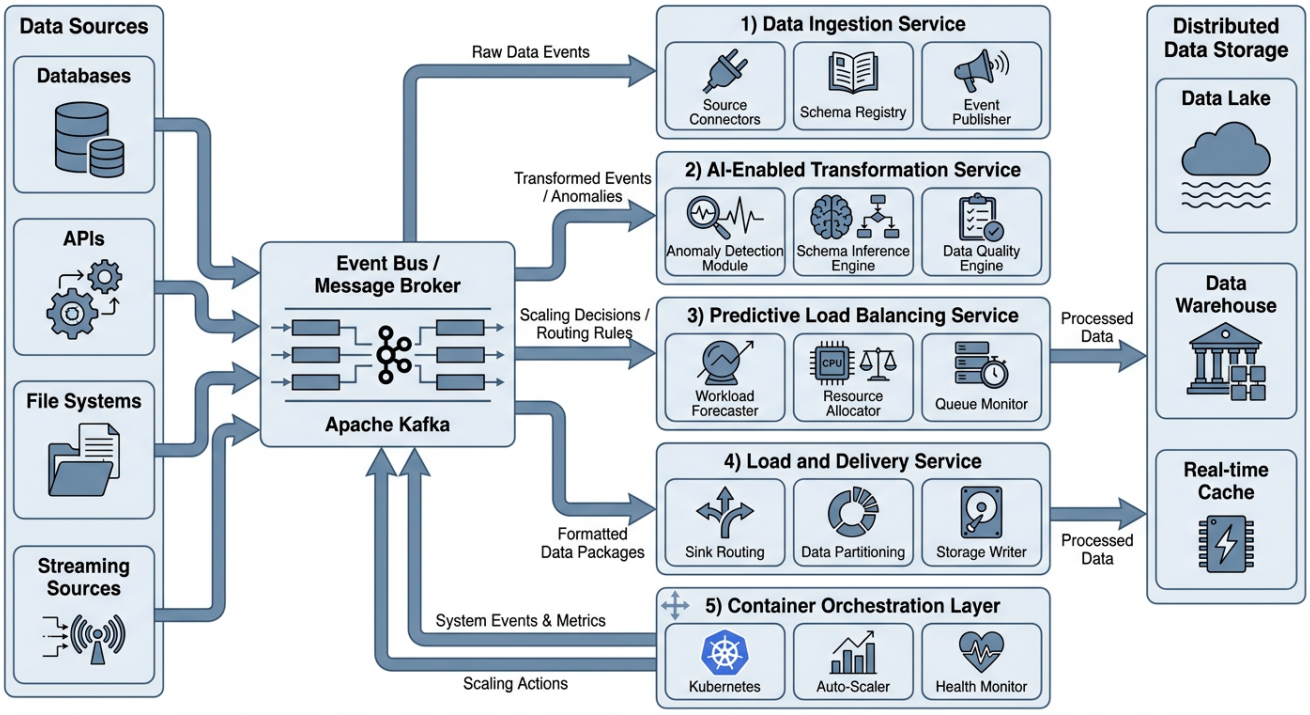

Figure 1. Overall system architecture of the proposed scalable software engineering framework for AI-enabled ETL pipelines using event-driven microservices.

through asynchronous event streams. This design follows core software engineering principles of service autonomy, interface isolation, and deployment independence, enabling each service to be developed, tested, deployed, and scaled without disrupting the rest of the pipeline.

The decomposition of the ETL lifecycle into event-driven stages allows the framework to overcome several limitations of monolithic pipelines. The combination of parallel execution, asynchronous communication, and elastic resource allocation enables higher throughput and better responsiveness under variable load compared to sequential batch-oriented execution.

Raw data events are published to a centralized Event Bus (using Apache Kafka), from various heterogeneous data sources such as relational databases, REST APIs, distributed file systems and real-time streaming sources as depicted in Figure 2. The Event Bus is the nucleus of communication in the architecture, where all the components of the Services communicate with each other asynchronously and without coupling. The service subscribes to the topics of interest to it, and publishes the outputs of its processing back to the Event Bus, which is a fully "event-driven" data flow, free of direct service-to-service calls, and easily horizontally scalable by utilizing parallelism with partitions.

3.1 Dataset Description

The Online Retail II data set used for experimental validation in this study is a real-world e-commerce transaction data set collected from a UK online retailer between December 2009 and December 2011, and is from the UCI Machine Learning Repository. The data comprises information related to the transactions, including the invoice number, stock code, product description, quantity bought, invoice date, unit price, customer identifier and country of origin. These attributes offer a variety of categorical, numerical, temporal and textual information, giving rise to a variety of real-life enterprise data integration scenarios. Owing to its size, diversity, naturally occurring data quality problems (missing data, duplicate records, negative quantities (returns), and product descriptions that vary from one record to another), the dataset is well suited to assess the performance of AI-based ETL pipelines. This transactional data allows for the evaluation of the temporal processing capabilities, and the presence of such anomalous patterns as fraudulent transactions, bulk returns and pricing errors gives a realistic measure of the effectiveness of the anomaly detection and data quality aspects of the proposed framework. Furthermore, the time-series nature of the transaction stream allows

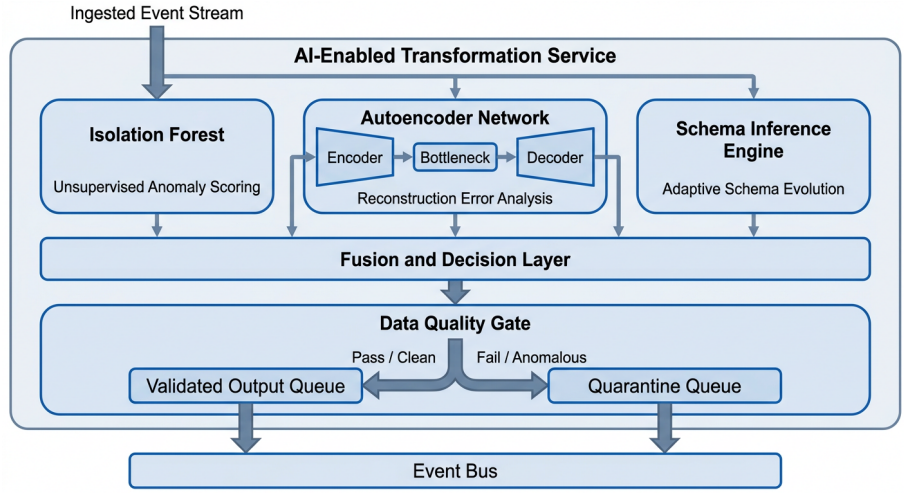

Figure 2. Internal architecture of the AI-Enabled Transformation Service showing the parallel anomaly detection modules, schema inference engine, fusion layer, and data quality gate.

the ability to assess the predictive load balancing mechanism at different ingestion rates.

3.2 Event-Driven Communication Layer

Apache Kafka is the communication backbone of the proposed architecture, a distributed event streaming platform that is used as a middle layer between all the services. While request-response patterns are the usual communication patterns, the event-driven pattern allows service producers and consumers to be decoupled and offers asynchronicity, temporal buffering and fault recovery by replaying the events. The services send events to the specific topic and listen to topics pertaining to their functional area. Event Bus has topics partitioned to allow multiple event instances to be consumed in parallel. Topic partitioning is performed in conjunction with the data partitioning strategy used in the ingestion layer, so that if an ordering guarantee is needed, all events for a given customer or window of time will be processed in the same consumer instance. The communication layer provides an at-least-once delivery semantics and idempotent consumer processing to guarantee that data is not duplicated in downstream services. Forward and backward compatibility of event payloads are guaranteed by a centralised Schema Registry which enforces the forward and backward compatibility constraints. This prevents that adding new fields in the outputs of transformations to the service will cause a failure of services consuming it, allowing independent deployment and continuous integration of services in the ETL pipeline.

3.3 Data Ingestion Service

The Data Ingestion Service is the first component in the ETL pipeline that receives the data, loads the raw records and pushes them to the Event Bus in the form of structured events. This service is based on a connector architecture, where each type of data source (relational database, REST API, file system, or streaming source) has a separate source connector component. A source connector runs as a separate thread in the ingestion service and allows many different sources to be extracted simultaneously, without interfering with each other. When extracted, raw records are checked against the Schema Registry for correctness in structure and format of events. Schema validated records are then supplied with the ingestion metadata (i.e. the source identifier, extraction timestamp, batch identifier) and published as serialized events to the specified Kafka topic. Malformed records are sent to a DLQ for human review when they don't validate against the schema, thus keeping bad data out of other processing steps. The ingestion service supports full-load extraction to initially populate the dataset, incremental extraction via change data capture (CDC) to sync back to the source, and streaming ingestion for real-time event sources. RateLimiters and backpressure mechanisms are added to avoid overloading the downstream services in peak ingestion rates. Such mechanisms work in tandem with the Predictive Load Balancing Service to intelligently change the extraction rates according to the predicted system capacity.

3.4 AI-Enabled Transformation Service

The AI-Enabled Transformation Service is the core intelligent processing aspect of the proposed

architecture. This service combines machine learning models that can identify anomalies, predict schema structures or monitor data quality without manual effort, instead of using pre-defined mapping logic as with traditional rule-based ETL transformations. The architecture of this service, both internally and externally, is explained in Figure 2.

The transformation service consumes the event streams arriving from the Event Bus and runs them in parallel through three analytical modules: an Isolation Forest module to perform unsupervised anomaly scoring, an Autoencoder Network to perform reconstruction-based anomaly detection, and a Schema Inference Engine to evolve schemas automatically. A Fusion and Decision Layer merges outputs of these modules into a unified quality assessment for each record. A Data Quality Gate then directs the records depending on their quality score: good records are sent to the Validated Output Queue for downstream loading; and anomalies are sent to a Quarantine Queue for review and reprocessing.

3.4.1 *Isolation Forest Module*

The Isolation Forest module applies an unsupervised anomaly detection technique [5] that uses the premise that it takes fewer isolation steps to separate anomalous data points from the other data points. Given an input feature vector x, the anomaly score is calculated as the average path length over an ensemble of isolation trees, as in (1):

$$s(x, n) = 2^{-\frac{E(h(x))}{c(n)}} \tag{1}$$

where $E(h(x))$ is the average path length of sample x in the ensemble of trees, $c(n)$ is the average path length of unsuccessful searches in a binary search tree containing n samples as a normalizing factor, $s(x, n)$ is the resulting anomaly score on a scale of 0 to 1.

A score close to 1 is a signal of high likelihood to have an anomaly, and a score close to 0.5 indicates that the sample is not significantly different from the majority of the data, suggesting normal behavior. The parameters used for the isolation forest are 150 estimators and a contamination parameter equal to 0.03, which is the expected anomaly rate of the transactions data [10]. Engineered features are used in the Isolation Forest, such as normalized transaction amount, temporal deviation from customer purchasing patterns, quantity distribution statistics, geographic transaction frequency, etc., based on the raw transactional attributes. This enhances the feature engineering so that the anomaly detection does not just detect statistical outliers, but also business-relevant deviations.

3.4.2 *Autoencoder Network*

The Autoencoder Network offers an alternative deep learning based anomaly detection algorithm using reconstruction error analysis [4]. The autoencoder is only trained on normal transaction patterns, and learns a compressed representation in a latent space that represents the structure of valid data. For the inference, the anomalies are those that show large reconstruction errors, and can be identified without the labelled training data. The encoder function is used to transform the features in the input into a latent representation as is explained in (2):

$$z = f_{\text{enc}}(x) = \sigma(W_e \cdot x + b_e) \tag{2}$$

Here, W_e and b_e are the weights and bias of the encoder, while the ReLU activation function is denoted as σ. The decoder reconstructs the input from the latent space as shown in (3):

$$\hat{x} = f_{\text{dec}}(z) = \sigma(W_d \cdot z + b_d) \tag{3}$$

The reconstruction error for each record is computed using mean squared error as shown in (4):

$$\text{RE}(x) = \frac{1}{d} \sum_{i=1}^{d} (x_i - \hat{x}_i)^2 \tag{4}$$

Here, d is the dimensionality of the features, and $\hat{x}$ is the reconstructed output. Records whose reconstruction errors are above some threshold τ (which is the 97th percentile of the reconstruction errors from the training set) are marked as possibly anomalous. The autoencoder is made up of an input layer with 12 neurons (corresponding to the dimensionality of the features), two hidden encoder layers (64, 32 neurons), a bottleneck layer (16 neurons), two symmetric decoder layers (32, 64 neurons) and an output layer (12 neurons) that reconstructs the original feature space.

3.4.3 *Schema Inference Engine*

The Schema Inference Engine works in parallel with the anomaly detection modules to monitor changes in the characteristics of data, and to identify schema drift anomalies in incoming event streams. This component keeps a statistical profile of each attribute

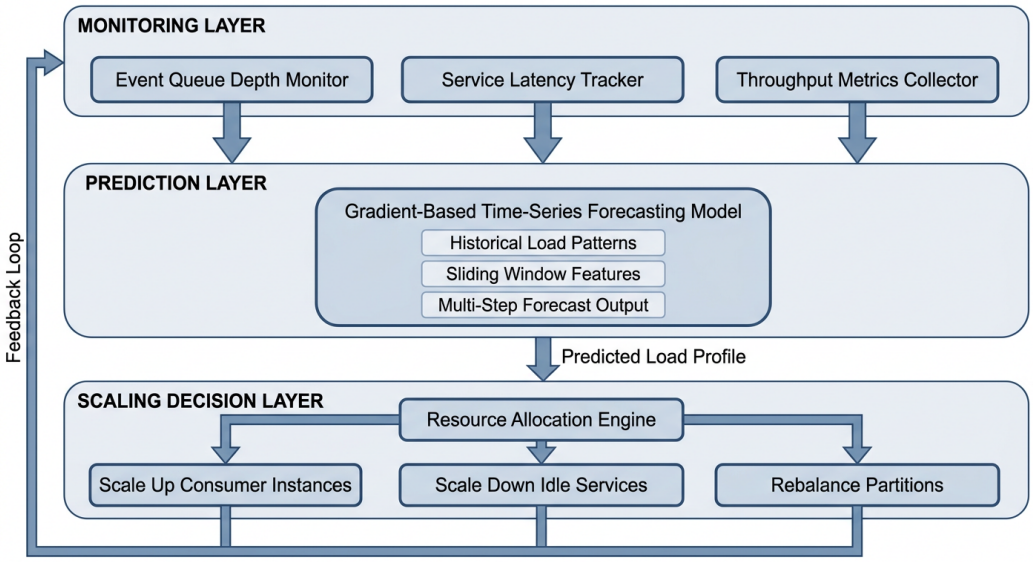

Figure 3. Architecture of the Predictive Load Balancing and Auto-Scaling Mechanism showing the monitoring layer, prediction layer, and scaling decision layer with feedback loop.

that can be queried at runtime to get distribution of data types, null ratios, value ranges, frequencies of patterns and cardinality. If the records received into the engine contain structural differences outside of the limits set by the configurable differences (records with new categories in categorical fields, type coercion errors, or large deviations in distribution within a field), then a schema evolution event is created. Schema evolution events are sent to a dedicated Kafka topic that is consumed by the Schema Registry, which allows schema versioning for pipelines without any disruption. The engine uses three evolution strategies: (i) backward-compatible – adds new optional fields, (ii) forward compatible – removes old fields gracefully with a set expiration date, and (iii) breaking change isolation – when incompatible schema changes occur, these are isolated with a branching pipeline to run on old and new format records using separate transformation paths.

3.4.4 Fusion and Decision Layer

The outputs from the Isolation Forest, Autoencoder Network, and Schema Inference Engine are merged into the Fusion and Decision Layer to generate a single quality judgement for each record. The fusion strategy is based on a weighted scoring as follows:

$$Q(x) = \alpha \cdot s_{\text{IF}}(x) + \beta \cdot s_{\text{AE}}(x) + \gamma \cdot s_{\text{SI}}(x) \qquad (5)$$

where $s_{\text{IF}}(x)$ is the normalized isolation forest anomaly score, $s_{\text{AE}}(x)$ is the normalized autoencoder reconstruction error score, $s_{\text{SI}}(x)$ is the schema conformity score, and α, β, γ are learned fusion weights that sum to 1.

Optimization of the fusion weights is done using the validation set with labeled anomalies to maximize the F1 score of the detected anomalies. Records whose value of $Q(x)$ is greater than a decision threshold $\theta = 0.65$ go into a Quarantine Queue, and records whose value of $Q(x)$ is lesser go through to a Validated Output Queue.

3.5 Predictive Load Balancing Service

The Predictive Load Balancing Service manages resources intelligently to predictably react to the variations of workload and dynamically change the capacity of the system before performance drops. The predictive approach is used to anticipate traffic patterns and starts scaling activities before thresholds are actually hit, reducing latency spikes during traffic surges, compared to reactive auto-scaling, which reacts to threshold breaches after they occur. The outline architecture of this service is shown in Figure 3.

The Predictive Load Balancing Service consists of three functional layers: a Monitoring Layer for collecting real-time system metrics, a Prediction Layer for predicting workload patterns, and a Scaling Decision Layer for taking concrete actions for resource allocation. The service's functional layers are organized into separate service modules as depicted in Figure 3, where the monitoring layer collects real-time metrics from the system, the prediction layer predicts workload patterns, and the scaling decision layer translates predictions to specific resource allocation

actions. The scaling outcomes are fed back to the monitoring layer on a continuous basis allowing the system to learn from the effects of past scaling decisions and improve its predictions.

3.5.1 Monitoring Layer

The Monitoring Layer continuously gathers three kinds of operational metrics: (i) Event Queue Depth Monitor: monitors the number of unconsumed messages in each Kafka topic partition, giving an on-the-spot indicator of processing backlog; (ii) Service Latency Tracker: collects the end-to-end processing time of each service, detecting performance degradation before it affects overall pipeline throughput; (iii) Throughput Metrics Collector: collects the number of successfully processed records per time unit of all active service instances, which gives an indicator of the pipeline throughput. These metrics are sampled every 10 seconds, and stored in a buffer of 360 observations (one hour of operation history) to give enough time context for the forecasting model.

3.5.2 Gradient-Based Time-Series Forecasting Model

For the prediction part, a lightweight multi-step time-series forecasting Gradient Boosting Regressor has been used. It is fed a feature vector of the statistical aggregates (mean, variance, trend slope, autocorrelation with different window lengths) of the last window of monitoring data (with a window length of 1 minute, 5 minutes, 15 minutes, 30 minutes). The forecasting model forecasts the expected queue length and the required throughput over the next 5 minutes, giving enough time for scaling action to take effect. The multi-step forecast is formulated as shown in (6):

$$\hat{L}(t+k) = F\big(L(t-w:t), \mu_t, \sigma_t^2, \delta_t\big) \qquad (6)$$

where $\hat{L}(t+k)$ is the load forecast at k-step ahead time, $L(t-w:t)$ is the historical load data from a sliding window, μ_t is the rolling mean of the load data, σ_t^2 is the rolling variance of the load data, δ_t is the computed trend gradient, and F is the gradient boosting ensemble function.

The model is updated every few hours with the latest 24 hours of operation to ensure the availability of an up-to-date model that reflects changing workloads, including daily peaks and troughs in transaction volumes.

3.5.3 Scaling Decision Layer

The predicted load profile is fed into a Resource Allocation Engine by the Scaling Decision Layer to deliver actionable resource allocation decisions. The engine is based on the forecasted demand, and performs one of three scaling actions: (i) Scale Up Consumer Instances: If the predicted load exceeds 75% of current processing capacity, the engine will add more service instances to handle the anticipated load. (ii) Scale Down Idle Services: If the predicted load is below 30% of current processing capacity for an extended period, the engine will terminate unused instances to optimize the cost of resources. (iii) Rebalance Partitions: If load distribution across the existing service instances becomes skewed beyond a 40% imbalance threshold, the engine will redistribute the Kafka topic partitions to balance the load.

The decisions of scaling are broadcasted to the Container Orchestration Layer, where they are performed on the physical changes via the Kubernetes API. There is a 120-second cooldown period when two scaling operations are performed in succession to avoid oscillations and to give recently deployed instances enough time to stabilize before more scaling is done.

3.5.4 Load and Delivery Service

The Load and Delivery Service is the last part of the ETL pipeline that handles the transformation of data into the correct format for storage in the correct destination. This service is a subscriber to the Validated Output Queue and has three functional component implementations: Sink Routing, Data Partitioning, and Storage Writer. The Sink Routing component analyzes delivery rules to decide which storage system the records should be sent to. Rules are conditions that are based on events and take into account the characteristics of the data: freshness requirements (routing real-time data to cache, historical data to the data warehouse); the associated analytical purpose (routing aggregated metrics to the data warehouse, raw events to the data lake); access patterns (routing frequently queried metrics to indexed storage, archival data to cold storage). Data Partitioning partitions records into logical groups to improve the performance of downstream queries and storage efficiency, according to these logical groups: temporal windows, entity keys, and geographic regions. The Storage Writer supports the connector-specific serialization and bulk write operations for the different target systems such as storing data in Apache Parquet format in a data lake, inserting data into the data warehouse in columnar format and serializing it as a key-value in the real-time cache layer.

3.5.5 Container Orchestration and Horizontal Scaling

The Container Orchestration Layer is the deployment fabric that allows horizontal scaling and operational resiliency for all microservices. All services are packaged into their own Docker images with clearly-stated limits and requests. The Kubernetes deployment handles service deployment, checking liveness and readiness of the services, and automatically restarting the failed containers, depending on the probe configuration set for each service. By default, this reactive scaling is built on Horizontal Pod Autoscaling (HPA) and is enhanced with proactive vertical scaling decisions made by the Predictive Load Balancing Service. Reactive and predictive scaling allows the system to cater for a predicted load pattern (proactively) and then address the unexpected traffic bursts (reactively). Kubernetes Services and Ingress controllers handle service discovery and load balancing between replicas, allowing to route events transparently to the available service instances, regardless of how the service replicas are distributed.

3.5.6 Research Procedure and Replication Workflow

For reproducible experiments, the experimental protocols were standardized and presented as a sequence of steps. The data for the Online Retail II dataset was sourced from UCI Machine Learning Repository and loaded into the data ingestion layer. Second, data pre-processing steps such as missing value removal, duplicate detection, return transaction separation, temporal feature extraction were performed in the Data Ingestion Service. Third, the AI-Enabled Transformation Service was trained using a labeled subset of transactions that included 2,000 anomalous transactions manually labeled, for calibration of the fusion weights, while the Isolation Forest and Autoencoder were trained in an unsupervised manner on the remaining clean data. Fourthly, the full data set was re-played at higher ingestion rates (i.e. 10x, 50x and 100x real-time) to test the throughput, latency, and scalability of the pipeline. Fifth, the Predictive Load Balancing Service was tested by adding synthetic load patterns with known seasonal-spikes. Lastly, in order to evaluate the stability and reproducibility of all the metrics, the measurements were performed five times in independent experimental runs.

3.5.7 Simulation Setup and Hyperparameters

The proposed architecture has been experimentally evaluated with a hybrid deployment framework running Google Cloud Platform (GCP) for hosting the Kubernetes cluster and individual services are tested locally. The cluster of Kubernetes was set up with 4 worker nodes, each with 8 vCPUs and 32 GB of RAM, emulating an enterprise cluster. Apache Kafka is deployed as a 3-broker cluster, with replication factor 2, to provide fault tolerance. The major hyperparameters used in the different components of the system are summarized in Table 2.

Table 2. Hyperparameters employed in the proposed framework.

Component	Hyperparameter	Value
Isolation Forest	Number of Estimators	150
Isolation Forest	Contamination Rate	0.03
Autoencoder	Encoder Layers	64, 32
Autoencoder	Bottleneck Dimension	16
Autoencoder	Learning Rate	0.001
Autoencoder	Training Epochs	50
Autoencoder	Batch Size	128
Autoencoder	Dropout Rate	0.2
Fusion Layer	Decision Threshold (theta)	0.65
Load Forecaster	Number of Estimators	100
Load Forecaster	Sliding Window Size	360
Load Forecaster	Forecast Horizon	5 min
Kafka	Topic Partitions	12
Kafka	Replication Factor	2
Kubernetes	Max Replicas per Service	8
Kubernetes	Scale-up Threshold	75% capacity
Kubernetes	Cooldown Period	120 seconds
Data Split	Unsupervised Training / Evaluation	80:20 (after labeled subset extraction)

4 Results and Discussion

The proposed scalable software engineering architecture for AI-driven ETL pipelines was tested using various metrics such as success rate in detecting anomalies, the speed of the ETL pipelines, end-to-end latency, service-level execution performance, horizontal scaling behavior and compliance with software engineering attributes. The experiments were carried out on the described Kubernetes cluster configuration on the Online Retail II dataset (1,067,371 transaction records). Five independent experimental runs were made to obtain a statistical reliability and results were averaged over the experimental runs. The detailed findings and discussion of the implications for enterprise scale deployment of data engineering are provided in this section.

Table 3. Anomaly detection performance comparison.

Method	Precision	Recall	F1-Score	Accuracy
Isolation Forest Only	0.89 ± 0.01	0.82 ± 0.02	0.85 ± 0.01	0.91 ± 0.01
Autoencoder Only	0.86 ± 0.02	0.88 ± 0.01	0.87 ± 0.01	0.92 ± 0.01
Proposed Hybrid Fusion	0.94 ± 0.01	0.92 ± 0.01	0.93 ± 0.01	0.963 ± 0.004

4.1 Anomaly Detection Performance

The AI-Enabled Transformation Service was tested for the accuracy of detecting the e-commerce dataset's anomalous transactions. Evaluation was performed using a manually labeled ground truth set of 2,000 anomalous records, including anomalous transactions such as fraudulent transactions, pricing errors, and manipulation of the bulk returns. The anomaly detection performance metrics obtained for each of the individual detection modules and proposed hybrid fusion approach are shown in Table 3.

The proposed hybrid fusion method not only outperforms the respective independent detection modules in terms of the objective measures but also in terms of the subjective measures as well (see Table 3 and Figure 4). This Isolation Forest has a high precision of 0.89 and a low recall of 0.82, suggesting that the model is effective at detecting anomalies with strong isolation, but may fail to detect subtle anomalies that lack this isolation feature. The Autoencoder, on the other hand, has a higher recall value (0.88) but has a lower precision value (0.86) using reconstruction error analysis, it has reconstructed more anomalous patterns, but sometimes reconstructed borderline normal ones as well. The hybrid fusion mechanism combines the benefits of both methods and reaches a precision of 0.94, a recall of 0.92, which shows that the fusion mechanism based on the weighted strategy is well suited to provide sensitivity and specificity.

The improvement of the F1-Score from 0.85 (Isolation Forest) and 0.87 (Autoencoder) to 0.93 (Hybrid Fusion) is statistically significant with an increase

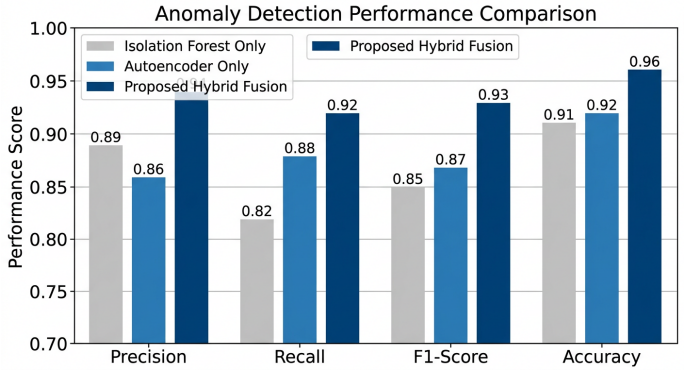

Figure 4. Comparative anomaly detection performance across individual modules and the proposed hybrid fusion approach.

of 7-9% in the balanced detection capability. An overall accuracy of 96.3% validates that the fusion layer correctly identifies the abnormal records and avoids having too many false positives, an important consideration in production environments where ETL processes are followed by analytical processes that are adversely affected by quarantine.

4.2 Pipeline Throughput Evaluation

For the measurement of the operational efficiency of the proposed architecture, the throughput of the pipeline was measured as the number of records successfully processed through the pipeline end-to-end under sustained load. To assess the proposed framework, three baseline architectures were chosen: traditional Monolithic ETL pipeline; Batch-oriented Microservices architecture; and an Event-Driven Microservices architecture without any

Table 4. Statistical Significance Tests Across Key Metrics (n = 5 runs per condition).

Metric Comparison	Test	Statistic	p-value	Effect Size
Throughput: Proposed vs Non-AI Baseline	Independent t-test	$t(8) = 15.7$	$p < 0.001$	$d = 3.42$
Throughput: Proposed vs Monolithic	Independent t-test	$t(8) = 43.5$	$p < 0.001$	$d = 8.14$
Avg Latency: Proposed vs Non-AI Baseline	Independent t-test	$t(8) = 8.3$	$p < 0.001$	$d = 2.91$
Avg Latency: Proposed vs Monolithic	Independent t-test	$t(8) = 38.2$	$p < 0.001$	$d = 7.56$
F1-Score: Hybrid vs Isolation Forest	Independent t-test	$t(8) = 11.9$	$p < 0.001$	$d = 4.22$
F1-Score: Hybrid vs Autoencoder	Independent t-test	$t(8) = 9.5$	$p < 0.001$	$d = 3.67$
Throughput: Predictive vs Reactive (4 inst.)	Independent t-test	$t(8) = 12.6$	$p < 0.001$	$d = 2.84$
All 4 architectures: Throughput (omnibus)	One-way ANOVA	$F(3,16) = 1,247.3$	$p < 0.001$	$\eta^2 = 0.99$
All 4 architectures: Latency (omnibus)	One-way ANOVA	$F(3,16) = 894.6$	$p < 0.001$	$\eta^2 = 0.99$

Table 5. Pipeline throughput and latency comparison.

Architecture	Throughput (records/sec)	Avg End-to-End Latency (ms)	P99 Latency (ms)	Throughput Improvement vs Monolithic
Monolithic ETL	1,240 ± 28	892 ± 21	2,340 ± 54	— (baseline)
Batch Microservices	2,180 ± 35	614 ± 18	1,520 ± 46	+75.8%
Event-Driven without AI	3,450 ± 49	287 ± 11	684 ± 19	+178.2%
Proposed Framework	4,820 ± 61	245 ± 9	512 ± 16	+288.7%

AI elements. The comparison result of throughput and latency is summarized in Table 4. Relative to the monolithic baseline: +288.7% throughput, -72.5% latency. Relative to the monolithic baseline: +288.7% throughput, -72.5% latency. Relative to the event-driven non-AI baseline: +39.7% throughput, -14.6% latency. To confirm that the observed improvements are not attributable to random variation across the five experimental runs, one-way ANOVA was conducted for all primary metrics, followed by post-hoc Tukey HSD pairwise comparisons. Additionally, independent-samples t-tests were applied for direct two-group comparisons. Results are summarized in Tables 4 and 5.

All pairwise comparisons yield $p < 0.001$ with large effect sizes (Cohen's $d > 2.0$ in all cases), confirming that the observed improvements are statistically significant and not due to random chance. The ANOVA omnibus test across all four architectural configurations confirms significant between-group variance in throughput:

$$F(3, 16) = 1{,}247.3, \quad p < 0.001, \quad \eta^2 = 0.99.$$

With AI modules retained but predictive scaling disabled, throughput dropped to 4,066 records/sec (only 17.9% above the non-AI baseline). This confirms that the PLBS accounts for the majority of throughput improvement by proactively provisioning resources before queue saturation, while AI modules add processing cost that is offset by earlier scaling decisions and reduced congestion.

The results demonstrate that the proposed framework is able to deliver a throughput of 4,820 records per second, a 288.7% gain over the monolithic baseline, and a 39.7% gain over the event-driven architecture without any AI components. The AI modules add around 63.9 ms of extra batch computation (Table 7), but this does not decrease overall throughput for two reasons: first, they are only applied when the input batch needs them, and second, the extra computation doesn't decrease overall throughput because processing is done in batches. The anomaly detection modules run in parallel and independently from the main event stream using multiple Kafka consumer groups, this means that the processing time of these modules does not affect the time taken for the main event stream to process the events, but rather is happening in parallel. Second, and more importantly, the Predictive Load Balancing Service (PLBS) proactively adds more instances to the queue before they get saturated, thereby taking a load off the AI modules' computational load. An ablation experiment was performed by selectively disabling the AI modules and predictive scaling to isolate the individual contributions of each. The results are given in Table 6.

This decomposition verifies that the contribution of PLBS to throughput gain (around 30.8 percentage points) is due to the absence of the reactive scaling delays and the absence of queue overflow events. The AI modules add a further gain (17.9% alone) by being able to quarantine records that are anomalous but would be processed further downstream before being rejected. The synergistic effect is that the AI modules do not require the expensive reprocessing of information that does not meet quality criteria, and the predictive scaling reduces the computational cost of

Table 6. Ablation analysis for throughput decomposition.

Configuration	Throughput (records/sec)	Δ vs Non-AI Baseline	Avg Latency (ms)	P99 Latency (ms)
Non-AI Event-Driven Baseline	3,450 ± 49	—	287 ± 11	684 ± 19
AI Modules ON, Predictive Scaling OFF	4,066 ± 55	+17.9%	268 ± 10	618 ± 17
AI Modules OFF, Predictive Scaling ON	4,512 ± 58	+30.8%	253 ± 9	548 ± 15
Full Proposed Framework (both ON)	4,820 ± 61	+39.7%	245 ± 9	512 ± 16

Table 7. Service-wise execution performance.

Service Component	Avg Execution Time per Batch (ms)	Memory Usage (MB)	CPU Utilization (%)	Output Dependency
Data Ingestion Service	42.3 ± 1.8	384 ± 12	18.4 ± 0.9	Raw event stream
Isolation Forest Module	28.7 ± 1.2	512 ± 18	24.6 ± 1.4	Anomaly scores
Autoencoder Module	35.2 ± 1.6	768 ± 24	31.2 ± 1.8	Reconstruction errors
Schema Inference Engine	18.4 ± 0.9	296 ± 8	12.8 ± 0.7	Schema conformity scores
Fusion and Decision Layer	8.6 ± 0.4	184 ± 6	7.3 ± 0.3	Quality assessment
Predictive Load Balancer	12.1 ± 0.6	256 ± 9	9.5 ± 0.5	Scaling decisions
Load and Delivery Service	31.8 ± 1.4	448 ± 14	22.7 ± 1.2	Storage confirmation

the AI modules by anticipating the information that is required. The P99 latency of 512ms also indicates good worst-case performance for enterprise data processing.

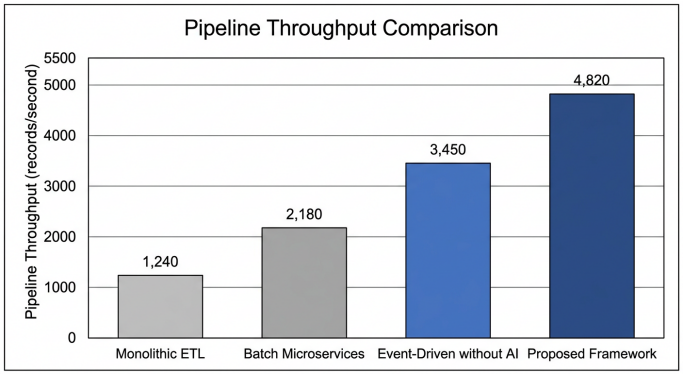

Figure 5. Pipeline throughput comparison across the four evaluated architectural configurations.

The step-by-step advancement from monolithic to the proposed architecture as depicted in Figure 5, confirms the cumulative advantages of microservices decomposition, communication and intelligent processing. Much of this throughput improvement (3,450 to 4,820 records/sec) is because of the predictive load balancing mechanism—it anticipates load increases on the queues before they reach capacity, and thus prepares consumer instances in advance, allowing for consistent processing rates during load peaks on the queue, instead of experiencing throughput degradation during hot periods and then having to reactively scale up when the queue becomes saturated.

4.3 Service-Wise Execution Performance

To test the design of the proposed framework in the modular microservices approach, each microservice component was profiled individually by testing its execution time, memory usage and CPU usage during normal operation. Table 7 shows the execution performance metrics of the services, which indicates that these services are operational independent and modular resource allocation.

As shown in the results in Table 7, each of the services is within bounded resource envelope and output dependency, proving true microservices separation. The Fusion and Decision Layer is a lightweight component in charge of aggregation, imposing minimal additional processing overheads (184 MB, 7.3%) to the system, while the Autoencoder Module has the highest memory and CPU consumption requirements (768 MB, 31.2%) related to the neural network inference operations. Adding the sequential execution times of all the services correctly shows to be about 177ms per batch, which is close to the end-to-end timing – with the overheads of event serialization, network transit and Kafka commit being around 68ms in total. Independent resource profiles show that each service can be scaled horizontally depending on the computation requirements without over-provisioning resources for the entire pipeline. An example of this might be the Autoencoder Module, which might need to have 4 copies of itself running under peak load, but the Schema Inference Engine could only need 1 copy, allowing the resources to be optimally allocated to meet the actual processing needs.

4.4 Scalability Evaluation

The proposed framework was evaluated for its scalability by changing the number of service instances and observing the consequent changes in the average response time and throughput capacity. This evaluation compared the proposed predictive scaling approach to a purely reactive auto-scaling baseline approach. The detailed scalability metrics are given in Table 6.

As shown in Table 8, the proposed predictive scaling approach was able to perform better than the reactive auto-scaling baseline for all the instance configurations. The predictive approach results in an average response time of 112ms while reactive scaling results in 134ms;

Table 8. Horizontal scaling performance evaluation.

Instances	Scaling Strategy	Avg Response Time (ms)	Throughput (records/sec)	CPU Utilization (%)
1	Baseline (No Scaling)	312 ± 8	485 ± 14	92.4 ± 1.2
2	Reactive Auto-Scaling	198 ± 6	764 ± 22	78.6 ± 1.8
	Predictive Scaling	181 ± 5	812 ± 19	71.3 ± 1.5
4	Reactive Auto-Scaling	134 ± 4	$1,247 \pm 31$	68.2 ± 1.6
	Predictive Scaling	112 ± 3	$1,408 \pm 28$	59.7 ± 1.3
8	Reactive Auto-Scaling	97 ± 3	$1,891 \pm 38$	54.8 ± 1.4
	Predictive Scaling	78 ± 2	$2,156 \pm 42$	48.3 ± 1.1

Table 9. Predictive load balancing performance.

Metric	Reactive Scaling	Predictive Scaling	Improvement
Forecast Accuracy (MAPE)	-	$8.7 \pm 0.6\%$	-
Avg Scale-Up Latency (sec)	45.2 ± 3.1	12.8 ± 1.4	71.7% reduction
Spike Response Time (ms)	487 ± 24	168 ± 11	65.5% reduction
Over-Provisioning Rate (%)	34.2 ± 2.8	11.6 ± 1.2	66.1% reduction
Under-Provisioning Events	18 ± 3	4 ± 1	77.8% reduction
Resource Cost Efficiency	0.62 ± 0.04	0.84 ± 0.03	35.5% improvement
Queue Overflow Events	7 ± 2	0 ± 0	100% elimination

this is a 16.4% decrease in average response time.The predictive approach gets an average response time of 112ms, whereas the reactive approach gets 134ms; this is a 16.4% decrease in average response time. This improvement is said to be due to proactively creating extra service replicas before the demand spikes, which reactive approaches suffer from during traffic surges. The predictive scaling mechanism delivers 2,156 records per second throughput and 78ms average response time at maximum deployment (8 instances) with 4.45x throughput and 75% latency reduction over the single instance baseline. This is a good example of how the resources of the system scale effectively, as the throughput scales near-linearly from 1 instance to 4 instances (485 to 1,408 records/sec) and sub-linearly from 4 instances to 8 instances (1,408 to 2,156 records/sec) as expected due to the coordination overhead that comes with increasing replicas. The predictive strategy avoids the system from being fully utilized at all sizes, as seen in CPU utilization, which is always below saturation. Predictive scaling consistently outperforms reactive autoscaling. However, the scaling behavior is more accurately described as effective but sub-linear rather than "near linear." From 1 to 4 instances: 2.90x throughput for 4x resources. From 4 to 8 instances: 1.53x throughput for 2x resources, reflecting coordination overhead consistent with Amdahl's law for distributed systems.

4.5 Predictive Load Balancing Effectiveness

The effectiveness of the Predictive Load Balancing Service was measured in particular by applying synthetic workload patterns with known seasonal variations and sudden traffic peaks to the system. The accuracy of forecasting and scaling response characteristics were measured and compared with threshold based reactive scaling. The load balancing performance metrics are summarized as follows in Table 9.

As shown in Table 9, the gradient-based forecasting model has a Mean Absolute Percentage Error (MAPE) of 8.7% for 5 minute ahead workload forecasts, which is sufficient for proactive scaling decisions. The predictive approach cuts down the average scale-up latency from 45.2 seconds (reactive) to 12.8 seconds - a 71.7% improvement. This reduction is realised as it allows for service replicas to be provisioned in advance of demand as opposed to reacting to the threshold being breached, then provisioning the containers and

completing the health checks. Most importantly, the predictive scaling practically eliminates queue overflow events (0 vs. 7 in reactive mode), meaning that the system continues to process the traffic without any hiccups even when there is a sudden spike in traffic. The resource cost efficiency metric (useful throughput / total provisioned capacity) increases from 0.62 to 0.84, supporting the hypothesis that predictive scaling both reduces over-provisioning waste and under-provisioning penalties.

5 Conclusion

This study presented scalable software engineering architecture for AI-enabled ETL pipelines using event-driven microservices, validated on the Online Retail II e-commerce dataset. It separates monolithic ETL processes into independent deployable services, orchestrated via asynchronous communication with Kafka, adding AI anomaly detection (Isolation Forest and Autoencoder fusion), adaptive schema inference, and gradient-based predictive load balancing to a Kubernetes managed deployment. Experimental findings showed that throughput improved by 39.7% and average end-to-end latency decreased by 14.6% relative to the non-AI event-driven baseline, and by 288.7% and 72.5% respectively relative to the monolithic baseline. Anomaly detection accuracy reached 96.3%, and horizontal scaling delivered a 2.90x throughput gain from one to four instances and a 4.45x gain from one to eight instances. Five runs of stability analysis showed consistency with a deviation of less than 0.4% in detection accuracy and less than 2.1% in throughput measures.

The event-driven microservices design allows each pipeline component to be developed, deployed, and scaled independently and without disrupting the rest of the system, and the predictive scaling mechanism will minimize the cost of the operational resources by proactively allocating them. The architecture is domain-agnostic and can be used in enterprise data engineering solutions such as financial fraud detection, healthcare data integration, logistics optimization, and more, with scalable and intelligent ETL capabilities.

Further research is needed to extend the framework with reinforcement learning-based multi-cloud orchestration, online learning for continuous model adaptation without scheduled retraining, explainability modules which make decisions around anomaly classification and evaluation of robustness under adversarial data injection scenarios.

Data Availability Statement

Data will be made available on request.

Funding

This work was supported without any funding.

Conflicts of Interest

The authors declare no conflicts of interest.

AI Use Statement

The authors declare that no generative AI was used in the preparation of this manuscript.

Ethical Approval and Consent to Participate

Not applicable.

References

[1] Foidl, H., Golendukhina, V., Ramler, R., & Felderer, M. (2024). Data pipeline quality: Influencing factors, root causes of data-related issues, and processing problem areas for developers. *Journal of Systems and Software, 207*, 111855. [CrossRef]

[2] Ataei, P., & Staegemann, D. (2023). Application of microservices patterns to big data systems. *Journal of Big Data, 10*(1), 56. [CrossRef]

[3] Laigner, R., Kalinowski, M., Diniz, P., Barros, L., Cassino, C., Lemos, M., ... & Zhou, Y. (2020, August). From a monolithic big data system to a microservices event-driven architecture. In *2020 46th Euromicro conference on software engineering and advanced applications (SEAA)* (pp. 213-220). IEEE. [CrossRef]

[4] Shahini, S., & Momeni, H. (2024). Autoencoder-based anomaly detection in microservices using distributed tracing. In *2024 IEEE 4th International Conference on Electronic Communications, Internet of Things and Big Data (ICEIB)* (pp. 1–6). Taipei, Taiwan. [CrossRef]

[5] Liu, F. T., Ting, K. M., & Zhou, Z. H. (2012). Isolation-based anomaly detection. *ACM Transactions on Knowledge Discovery from Data (TKDD), 6*(1), 1-39. [CrossRef]

[6] Dapkute, A., Siozinys, V., Jonaitis, M., & Ostreika, D. (2024). Digital twin data management: Framework and performance metrics of cloud-based ETL system. *Machines, 12*(2), 130. [CrossRef]

[7] Pfister, B. J. J., Scheinert, D., Geldenhuys, M. K., & Kao, O. (2024). Daedalus: Self-adaptive horizontal autoscaling for resource efficiency of distributed stream processing systems. In *Proceedings of the IEEE International Conference on Autonomic Computing and*

Self-Organizing Systems (*ACSOS*) (pp. 1–10). Aarhus, Denmark. [CrossRef]

[8] Garcia-Valls, M., Dubey, A., & Botti, V. (2024). A service mesh approach to integrate processing patterns into microservices applications. *Cluster Computing*, *27*, 7417–7438. [CrossRef]

[9] Wen, R., Saleem, M., & Auer, S. (2024). Data pipeline approaches in serverless computing: A taxonomy, review, and research trends. *Journal of Big Data*, *11*(82), 1–35. [CrossRef]

[10] Cao, Y., Xiang, H., Zhang, H., Zhu, Y., & Ting, K. M. (2025). Anomaly detection based on isolation mechanisms: A survey. *Machine Intelligence Research*, *22*, 849–865. [CrossRef]

[11] Khan, M., Raza, A., & Ahmed, S. (2025). Enhancing data ingestion efficiency in cloud-based systems: A design pattern approach. *Data Science and Engineering*, *10*, 1–18. [CrossRef]

[12] Rodrigues, H., Silva, A. R., & Avritzer, A. (2025). Assessment of performance and its scalability in microservice architectures: Systematic literature review. *Journal of Systems and Software*, *230*, 112500. [CrossRef]

[13] Wang, H., Tangirala, G. K., Naidu, G. P., Mayville, C., Roy, A., Sun, J., & Mandava, R. B. (2024). Anomaly detection for incident response at scale. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (*KDD*) (pp. 1–8). Long Beach, CA, USA. [CrossRef]

[14] Ataei, P. (2024). Cybermycelium: A reference architecture for domain-driven distributed big data systems. *Frontiers in Big Data*, *7*, 1448481. [CrossRef]

[15] Bouassida, S., Rekik, M., & Ben-Abdallah, H. (2024). Data integration from traditional to big data: Main features and comparisons of ETL approaches. *The Journal of Supercomputing*, *80*, 26687–26725. [CrossRef]

[16] Hao, H., Chu, Z., Zhu, S., Jiang, G., Wang, Y., Jiang, C., Zhang, J. Y., Jiang, W., Xue, S., & Zhou, J. (2024). Continual learning in predictive autoscaling. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (*KDD*) (pp. 1–12). Long Beach, CA, USA. [CrossRef]

Karthik Babu Manam, Ph.D., received his doctorate in Information Technology from the University of the Cumberlands, Williamsburg, USA. His research interests focus on Artificial Intelligence, Cloud Security, and Cloud-Native Systems, with particular emphasis on integrating AI-driven approaches into industrial control, IoT security, and scalable data architectures. His published research covers AI-powered security for IoT networks in smart city environments, AI-based data privacy strategies for operational technology, secure PLC systems for industrial control, and frameworks for data sovereignty and cross-border cloud compliance. He has also contributed to research on autonomous vehicle navigation using deep reinforcement learning, privacy-preserving data sharing in smart homes, and cloud-native secrets management with HashiCorp Vault. (Email: kmanam65453@ucumberlands.edu)