HyScaleFlow: An ML-Driven DAG-Based Orchestration Framework for Real-Time Stream Processing in Hybrid Cloud Environments

Srinivas Lakkireddy
Independent Researcher, USA
E-mail: reachlakkireddy@gmail.com

Keywords: hybrid cloud orchestration, real-time stream processing, machine learning, fault tolerance, resource optimization

Received: June 1, 2025

The increasing complexity of real-time data processing across hybrid cloud and edge environments has revealed significant limitations in existing distributed stream processing systems. While frameworks like Apache Spark and Flink offer strong scalability and performance, they lack the orchestration intelligence required to adapt to dynamic workloads, anticipate failures, and optimize resource usage in heterogeneous environments. Traditional rule-based or reactive orchestration approaches fail to deliver the responsiveness and fault resilience needed for mission-critical applications in domains such as IoT analytics, innovative infrastructure, and cyber-physical systems. To address these challenges, this paper presents HyScaleFlow, a scalable and modular framework that integrates real-time stream processing with machine learning–driven orchestration. The architecture combines Apache Spark (at the edge) and Apache Flink (in the cloud) with a hybrid DAG-based orchestration strategy using Apache Airflow and Dagster. A key innovation is the FlowGuard module, which uses XGBoost models (classifier and regressor) to predict node failures and forecast resource load based on Prometheus-exported telemetry metrics. These predictions dynamically inform DAG execution, enabling preemptive scaling, container migration, and workload-aware task routing. Evaluations were conducted using the NYC Taxi Trip dataset (over 1.1 billion records) on a hybrid cloud testbed that combines Spark at the edge and Flink in the cloud, orchestrated via Docker/Kubernetes. Results reveal that HyScaleFlow improves DAG completion rates by 16.8%, reduces task retry rates by over 60%, and enhances fault recovery times by up to 40%. Additionally, the framework achieves a 19.5% reduction in cloud execution cost and a 35.9% gain in resource efficiency. HyScaleFlow demonstrates strong utility for real-time, data-intensive applications by unifying predictive intelligence with stream processing. It provides a replicable, cost-effective, and resilient solution for hybrid cloud data engineering, advancing the state of intelligent orchestration.

Povzetek: Študija skuša omogočiti zanesljivo, samoprilagodljivo in stroškovno učinkovito obdelavo podatkovnih tokov v realnem času v hibridnih oblačno-robnih okoljih, z avtomatskim zaznavanjem anomalij, prerazporejanjem virov in preprečevanjem odpovedi za kritične industrijske in poslovne aplikacije. HyScaleFlow je hibridni okvir za sprotno obdelavo tokov: Spark na robu, Flink v oblaku, orkestracija Airflow+Dagster. Modul FlowGuard (XGBoost, Prometheus metrike) napoveduje odpovedi/nalaganje, sproži skaliranje/migracije.

1 Introduction

The explosive growth of real-time data generated by IoT devices, cloud applications, and cyber-physical systems has led to an increased adoption of distributed stream processing frameworks, such as Apache Spark and Apache Flink, in hybrid cloud environments. These frameworks offer high-throughput, low-latency processing but lack intelligent orchestration capabilities to adapt to unpredictable workloads, resource constraints, and system faults. Traditional orchestration strategies are primarily static, rule-based, or reactive, which limits their ability to ensure service continuity and efficiency in dynamic runtime conditions [1], [3]. Existing literature highlights the importance of autoscaling and stream framework benchmarking in hybrid deployments [12], [13], but few solutions integrate predictive machine learning with distributed data engineering pipelines. Moreover, most approaches do not coordinate multi-engine deployments across edge-cloud nodes or adapt DAG execution in real time based on system health metrics [14].

To address these limitations, this research proposes HyScaleFlow, a scalable, ML-enhanced framework for real-time data engineering and orchestration across hybrid cloud infrastructures. The primary objective is to design a modular system that enables predictive failure mitigation, workload-aware scaling, and

efficient task distribution using Apache Spark and Flink, orchestrated through Airflow, Dagster, and a novel ML module called FlowGuard. The key novelties of this research include: (i) integration of dual-stream processing with hybrid DAG orchestration, (ii) FlowGuard's real-time failure and load prediction using XGBoost models trained on Prometheus-exported metrics, and (iii) dynamic task routing and container management across edge and cloud nodes. These innovations enable intelligent orchestration beyond static or reactive models, supporting fault resilience, throughput efficiency, and operational cost reduction.

The contributions of this paper are threefold: first, it presents a robust, predictive orchestration architecture unifying multiple execution engines; second, it demonstrates significant improvements in execution metrics such as DAG completion, task retry rates, and system uptime through experimental validation; third, it offers a replicable deployment strategy supported by public datasets and open-source tools, enabling broader adoption in industry and academia.

In alignment with the proposed framework and its objectives, this study addresses the following research questions:

RQ1: Can a machine learning–driven orchestration strategy improve DAG completion and reduce task retry rates in hybrid cloud environments?

RQ2: How accurately can system-level telemetry metrics forecast node failure and workload surges using XGBoost-based predictive models?

RQ3: To what extent can predictive orchestration reduce cloud resource costs and improve throughput efficiency compared to rule-based alternatives?

These questions guide the design, implementation, and evaluation of HyScaleFlow and form the basis for the comparative experimental analysis presented in this paper.

The rest of this paper is organized as follows. Section 2 reviews related work in hybrid stream processing, orchestration strategies, and ML-driven system adaptation. Section 3 details the architecture, FlowGuard algorithm, and orchestration workflow in HyScaleFlow. Section 4 presents the experimental setup, performance evaluation, and visualization of results. Section 5 discusses the findings on existing works and outlines the system's limitations. Finally, Section 6 concludes the study and provides directions for future enhancements to increase generalizability, efficiency, and scalability.

2 Related work

This literature review explores scalable distributed data processing, hybrid cloud orchestration, and intelligent stream analytics using AI-enabled frameworks. Ullah et al. [1] compared Hadoop, Spark, and Flink on a hybrid cloud; Flink was the fastest, and Spark the most cost-effective. In the future, cross-cloud latency and scaling may be optimized for improved performance. Ponnusamy and Gupta [2] investigated the scalability and effectiveness of data partitioning in cloud processing; future research might enhance tactics for real-time cloud analytics. Henning and Hasselbring [3] scaled benchmarks for stream frameworks, revealing linear scaling but varying efficiency; further research may optimize cost-performance trade-offs. Irshad et al. [4] proposed a secure IoT-cloud connection utilizing an SSCA that incorporates MBRA, PQC, and blockchain, with performance verified. Further development would include broader scalability. Islam and Bhuiyan [5] proposed a scalable green IoT-cloud healthcare platform that utilizes hierarchical clustering and does not validate energy measurements; further sustainability research will be explored in future studies.

Banimfreg [6] suggested cloud infrastructure for bioinformatics, and the present advantages were assessed. Drawbacks included privacy issues with data. Future work included enhancing security and training. Lohitha and Pounambal [7] employed push-pull and publish/subscribe communications; the proposed scalable IoT-cloud architecture reduces device overhead and may improve efficiency in the future. Singh et al. [9] performed better than other databases when evaluated against databases for financial time-series in a hybrid cloud; further research should examine larger datasets and latency measures. Khriji et al. [10] proposed that REDA is an inexpensive, real-time, event-driven IoT cloud system that utilizes Kafka and MQTT; further development may improve scalability.

Chen et al. [11] utilized NVMs to optimize Big Data memory utilization, thereby saving energy; further research can enhance flexibility across a range of workloads. Razzaq et al. [12] enhanced their approach with a hybrid burst-aware auto-scaling method; further research may improve real-time burst prediction and scalability cost-efficiency. Radhika and Sadasivam [13] examined hybrid auto-scaling tactics, emphasizing the difficulties in dynamic resource estimation and proposing proactive-reactive adaptive methods. Alsboui et al. [14] highlighted the main obstacles, categorized and examined distributed intelligence in the Internet of Things, and suggested future adaptive hybrid DI solutions. Risco et al. [15] demonstrated private smart city video processing using a hybrid serverless platform for elastic scientific operations.

Hu et al. [16] proposed a real-time traffic tile generation technique based on Apache Flink, which enhances the scalability and visualization performance of Intelligent Transportation Systems (ITS). Mohyuddin and Prehofer [17] offered a practical

framework for processing data from autonomous vehicles and evaluating driving behavior that is scalable and based on Spark. Rao et al. [18] suggested utilizing Spark and Flink to mine top-k user communities in weighted bipartite graphs in a distributed manner. Dongen and Poel [19] highlighted recovery times and semantics when assessing fault tolerance in Spark, Flink, Structured Streaming, and Kafka Streams. Ashiku et al. [20] investigated the use of Apache Spark for healthcare big data analytics and machine learning in effective organ distribution.

Mostafaei et al. [21] examined and suggested fixes for performance reduction in large data analytics systems (Storm, Spark, and Flink) caused by geographical delays. Shaikh et al. [22] extended Apache Flink to manage geographic data streams and enable spatial queries. GeoFlink outperforms existing platforms in terms of performance. Chen et al. [23] gathered and diagnosed escalator operating data using fault tree analysis and big data techniques (Flume, Kafka, Flink). Mostafaei et al. [24] suggested optimizing worker node placement in geo-distributed stream processing systems based on additive weighting. Almeida et al. [25] focused on strategies for managing large amounts of data and forecasts, while also discussing the development of real-time systems for analyzing big data.

Kastrinakis and Petrakis [26] used Flink for speed and Apache Kafka for real-time and constrained video processing. Video2Flink is a scalable solution. Chen et al. [27] examined use examples in finance and health, real-time analytics, and AI integration, emphasizing obstacles and potential paths forward. Xu et al. [28] suggested a Spark-based parallel AC automation technique for effective DNS log processing with faster matching. Using Spark, Mallik et al. [29] created a parallel fuzzy C-median clustering algorithm for massive data with enhanced scalability and accuracy. Hassan [30] examined big data technology and compared the ability of the ARIMA and Weibull BMTD models to predict internet congestion.

Berberi et al. [31] assessed 16 MLOps products and provided insights on efficient AI infrastructure and a strategy for choosing scalable platforms. Zeydan and Bafalluy [32] identified gaps in applying data engineering advancements to the telecom industry and made suggestions for future development and early adoption. Shahid et al. [33] examined cloud fault tolerance strategies, categorizing them as Reactive, Proactive, and Resilient, and emphasized the importance of AI in recovery. Karthikeyan et al. [34] proposed the SALDEFT method to reduce transmission overhead and energy consumption while enhancing fault tolerance in cloud computing. Alaei et al. [35] suggested an IDE and ANFIS-based adaptive fault detection technique for better fault tolerance and cloud computing workflow scheduling.

Table 1: Literature review summary of comparable works related to hyscaleflow

Reference	**Methodology**	**Key Findings**	**Key Findings**	**Limitations / Research Gap**	**Relevance to HyScaleFlow**
Ullah et al. [1]	Benchmarking Hadoop, Spark, and Flink in a hybrid cloud	Flink is fastest; Spark is cost-effective (total time = 2998 sec & efficiency score = 0.53)	Flink is fastest; Spark is cost-effective	No orchestration or ML integration	Validates engine selection for hybrid processing
Henning & Hasselbring [3]	Microservice-based stream processing evaluation	Shows linear scaling of cloud-native frameworks	Shows linear scaling of cloud-native frameworks	Ignores hybrid/cloud-edge deployment and orchestration	Highlights the need for hybrid DAG orchestration
Razzaq et al. [12]	Predictive auto-scaling using burst modeling	Improves cloud workload efficiency (accuracy 92 %)	Improves cloud workload efficiency	Lacks DAG orchestration and edge processing	Inspires FlowGuard's predictive scaling logic
Radhika & Sadasivam [13]	Statistical auto-scaling for cloud applications	Demonstrates dynamic resource adaptation	Demonstrates dynamic resource adaptation	No feedback-based orchestration or DAG intelligence	Supports ML-based adaptation in HyScaleFlow

Alsboui et al. [14]	Survey of distributed intelligence in IoT	Highlights the architectural flexibility of edge-cloud	Highlights the architectural flexibility of edge-cloud	No empirical orchestration evaluation	Aligns with system-level distribution in HyScaleFlow
Henning et al. [8]	Configurable stream benchmarking at scale	Proactive load balancing using prediction (accuracy = 92 %)	Offers tuning for stream workloads	Does not explore ML-driven orchestration paths	Justifies the need for adaptable orchestration layers
Shahid et al. [33]	Survey of cloud fault-tolerance techniques	Offers tuning for stream workloads (load (msg /sec) = 50000 - 500000)	Categorizes proactive vs. reactive models	No implementation of predictive recovery	Supports FlowGuard's fault prediction and DAG recovery logic
		Categorizes proactive vs. reactive models			

Nalini and Khilar [36] proposed using Reinforced Ant Colony Optimization (RACO) to schedule tasks in cloud computing more effectively, resulting in a 60% performance increase. Rehman et al. [37] discussed cloud computing fault-tolerance tactics, proactive and reactive techniques, frameworks, and future research objectives. Taraghi et al. [38] introduced LLL-CAdViSE, a cloud-based platform that addresses several experimental factors for assessing low-latency live video streaming. Fragkoulis et al. [39] examined the development of stream processing systems, emphasizing fault tolerance, flexibility, and data management, and discussed potential future developments. Ching et al. [40], with future development potential, AgileDart enhances edge stream processing by adapting to changing circumstances, thereby increasing reliability, scalability, and latency. Guan [42] proposed a hybrid cloud workflow scheduling procedure supplemented with a Levy-optimized Slime Mould Algorithm (SMA), which addresses both efficiency and security challenges in dynamically resourced cloud systems. Our approach significantly outperforms a basic implementation, enhancing task allocation, execution reliability, and network resilience, leading to more secure and optimized hybrid cloud infrastructures. Ilias et al. [43] On the other hand, concerning cryptographic progress in the context of secure cloud communication, the authors proposed an integrated framework using the new post-quantum cryptographic primitives HEDT. The paper contributes to cloud data security and key exchange mechanisms by providing quantum-resistant cloud solutions that protect the reliability of encrypted data transmission over cloud systems against quantum computing, thereby reinforcing the independence of the cloud data and its key exchange mechanisms. Tang et al. [44] centered on predictive modeling for Industrial IoT systems, presenting a hybrid deep learning architecture that fuses Long Short-Term Memory (LSTM) networks and Transformer models. Their algorithm boosts energy management system stability and accurately predicts the state of health (SoH) and charge for battery management. This model performs with high precision and versatility, which is crucial for industrial real-time IoT applications.

Table 1 summarizes key literature relevant to HyScaleFlow, highlighting their methodologies, findings, limitations, and how they collectively inform the framework's design and research contributions. The review spans over 40 references covering performance comparisons of Spark and Flink, fault-tolerant orchestration, real-time stream optimization, and hybrid cloud innovations. Several works emphasize adaptive autoscaling, ML-based orchestration, and geo-distributed processing, while others focus on energy efficiency, IoT integration, and future-ready AI-enhanced orchestration strategies in hybrid environments.

3 Proposed framework

The proposed HyScaleFlow framework integrates predictive intelligence with dynamic orchestration to address the challenges of real-time, distributed data processing in hybrid cloud environments. It combines Apache Spark and Apache Flink for edge and cloud stream processing, utilizing a hybrid DAG orchestration mechanism that leverages Airflow and Dagster. The core intelligence module, FlowGuard, leverages XGBoost models to forecast failures and workload surges, enabling preemptive task migration,

adaptive scaling, and enhanced system efficiency across heterogeneous execution environments.

3.1 System overview

The HyScaleFlow framework is designed as a modular, distributed data engineering system capable of executing real-time processing pipelines across a hybrid cloud infrastructure. It seamlessly integrates ingestion, processing, orchestration, and intelligent decision-making to handle high-velocity data streams with scalability, reliability, and adaptability. The system leverages edge and cloud computing environments to optimize latency and resource availability while ensuring continuous data flow and pipeline resilience. Figure 1 presents a high-level architectural view of the entire HyScaleFlow system, illustrating the interaction between its key components.

The pipeline begins with external data sources such as the NYC Taxi Trip dataset [41], which emits timestamped records. These are ingested in real-time via Apache Kafka, acting as the primary message broker and buffer. Kafka partitions the incoming stream based on configured keys, supporting parallel processing. From Kafka, the data is streamed simultaneously to both edge and cloud nodes. The edge node hosts Apache Spark for latency-sensitive batch and stream tasks, while the cloud node runs Apache Flink for high-throughput, event-driven stream analytics. This architecture enables HyScaleFlow to handle diverse analytics requirements. The edge node processes latency-sensitive tasks, while the cloud node manages high-throughput analytical workloads. Together, they support distributed deployments with greater flexibility and resilience.

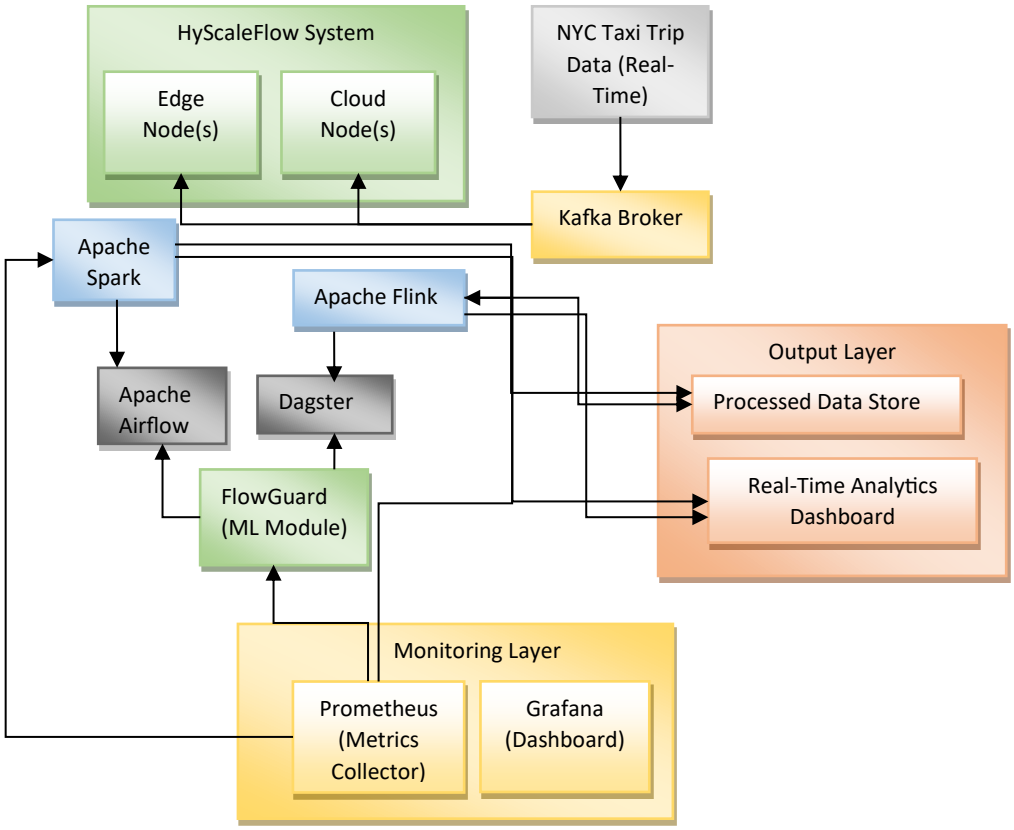

Figure 1: HyScaleFlow system architecture for real-time distributed data engineering in hybrid cloud

A hybrid strategy integrating Apache Airflow and Dagster handles workflow orchestration within the system. Airflow manages high-level DAG scheduling and periodic task triggering, whereas Dagster supports dynamic, type-aware execution paths and task retries based on data state and system feedback. FlowGuard, the embedded ML module, informs the orchestration layer, which receives real-time system health metrics from Prometheus. Based on its predictions, FlowGuard issues orchestration triggers that dynamically adapt the DAG execution, scale task branches, or migrate containers across nodes.

Processed results are streamed into distributed object storage systems or visualized in real time through a Grafana dashboard. This feedback loop enables continuous monitoring and fine-grained observability of system components, execution paths, and orchestration outcomes. The system design prioritizes modularity, extensibility, and real-time adaptability, making HyScaleFlow suitable for complex hybrid cloud deployments where performance and fault tolerance are critical. Table 2 defines key symbols and variables used throughout the HyScaleFlow

framework, including data streams, processing rates, and predictions.

Table 2: Notations used in the hyscaleflow framework, covering symbols related to data streams, processing metrics, orchestration logic, and ml-based prediction models

Notation	Description
x_t	Data record (event) at time t
$S = \{x_1, x_2, \ldots, x_t\}$	Input data stream as a sequence of events
$T(x_t)$	Timestamp extraction function for event x_t
$W(t)$	Watermark function to handle late data
B_i	Micro-batch of events in Spark for time interval $[t_i, t_i + \Delta t)$
$f(B_i)$	Transformation function applied on Spark batch B_i
$g_k(x_t)$	Flink function applied to event $x_t$with key k
$h(x_t)$	Key extraction function for partitioning in Flink
λ_{in}	Ingestion rate from Kafka into the system
λ	Effective workload rate per node
λ_{proc}	Processing rate of downstream engines (Spark/Flink)
L_{ingest}	Ingestion latency (difference between consumption and production timestamps)
C	Per-node processing capacity
p	Number of parallel executors or task managers
$G = (V, E)$	Directed Acyclic Graph for task orchestration (nodes V, edges E)
T_i	Individual task node in the DAG
$dep(T_i)$	Set of upstream tasks dependent for T_i's execution
R_i	Runtime status of task T_i (e.g., success, fail)
$x_t \in \mathbb{R}^d$	Feature vector at time t for FlowGuard input
x'_{ij}	Standardized value of feature j at sample i
$\hat{y}_t$	Predicted failure probability from FlowGuard (binary classifier output)
τ	Threshold for failure alert trigger (e.g., 0.7)
$\hat{r}_{t+1}$	Predicted resource usage for next time step from regression model

3.2 Data ingestion and streaming pipeline

HyScaleFlow handles data ingestion and streaming pipelines, starting with continuously sourcing high-velocity data from outside sources like the NYC Taxi Trip dataset, an example of timestamped, hectic geospatial and transactional data entries. The records are produced near or simulated near real-time and are inputted into the system via a distributed message broker—in our case, Apache Kafka. Kafka, as the first buffer layer that creates the data separation between producers and processing engines, guarantees that streams are transmitted in a fault-tolerant, sequenced, and scalable manner.

We model each incoming data record x_t at timestamp t as a tuple of structured attributes like pickup time, drop-off location, passenger count, and fare value. An event stream can be defined as $S = \{x_1, x_2, \ldots, x_t\}$,,where each $x_t \in \mathbb{R}^d$ correspond to a vector in d -dimensional feature space. These records are partitioned into different Kafka topics with some key stuff (vendor ID, pickup zone, etc.) so that they can be processed in parallel streams. Kafka handles at-least-once delivery guarantees and offset tracking for stream replay on failure.

Connector APIs: Connector APIs are used by both Apache Spark and Apache Flink to consume the Kafka stream. For event-time processing, every consumer consumes data from a partitioned topic and finds a timestamp extraction function, $T(x_t) \rightarrow t$. Watermarking strategy $W(t)$ is introduced to deal with the out-of-order events by providing the system with a threshold of maximum delay. That one is x_t considered late if $t < W(t)$,, which is used to drop or reroute stale inputs to different queues.

Let us denote the throughput of the ingestion layer as λ_{in}, and the effective consumption rate of the processing engines as λ_{proc}. Thus, to prevent the backlogs from accumulating, it must hold that the system should maintain that to satisfy Eq. 1.

$$\lambda_{proc} \geq \lambda_{in} \quad (1)$$

If those constraints are violated, it indicates a potential bottleneck, which triggers alerts and adjustments via FlowGuard. The system also monitors latency using Eq. 2.

$$L_{ingest} = t_{consume} - t_{produce} \quad (2)$$

Where $t_{produce}$ is the Kafka publish timestamp, and $t_{consume}$ is the timestamp when the record is read by the consumer One of the core input features in flowguard's predictive model is this latency metric. In summary, the ingestion and streaming pipeline in HyScaleFlow delivers timestamp-aligned, reliable, and parallel inflow of data to a hybrid cloud environment, which serves as the building blocks for scalable distributed data engineering.

3.3 Distributed processing in hybrid cloud nodes

We propose a design of distributed hybrid cloud execution nodes in the HyScaleFlow framework, where different execution nodes could be set up so that the latency-sensitive tasks can be distributed on the edge infrastructure. In contrast, the cloud environments can be used to maximize the computational power of mass-scale operations. New data streams from the Kafka broker get routed in real-time to both the edge nodes with Apache Spark and the cloud nodes with Apache Flink. This two-pronged approach allows for both real-time local processing and aggregate stream processing, with the ability to react and scale.

This edge node runs Spark Structured Streaming jobs in which streaming data is divided into small time intervals. Δt that form the micro-batches. Events within a micro-batch B_i are such that $\{x_{i1}, x_{i2}, \dots, x_{in}\}$ their timestamps fall within the interval $[t_i, t_i + \Delta t)$. Given a transformation function f, the outcome of batch processing is defined as in Eq. 3.

$$y_i = f(B_i) = f(\{x_{i1}, x_{i2}, \dots, x_{in}\}) \quad (3)$$

These outputs are checkpointed to HDFS for fault recovery and job replay capability. With its query execution engine, Spark guarantees stateful stream processing with exactly-once semantics.

Meanwhile, the cloud node deals with the stream simultaneously using Apache Flink, which works on an event-at-a-time basis in a very fine-grained state. Every records x_t coming is processed right away and stored in keyed state backends. Assume that the keyed function $g_k(x_t)$ is a transformation on record x_t with key k, then it can be expressed as in Eq. 4.

$$z_t = g_k(x_t), \quad \text{where } k = h(x_t) \quad (4)$$

Where $h(x_t)$ is the function for extracting keys A function g_k might even consist of aggregations, say windowed sums or joins, or pattern matching on event streams. Flink uses watermarking policies $W(t)$ (c.f. ingestion layer) to trigger processing windows and deal with late events.

Operator-level parallelism is preserved to scale across task slots for both Spark and Flink. Now, let us denote the number of executors/task managers p by, and the workload rate per node by λ. Total per-node capacity C which must hold on to condition in Eq. 5 to keep processing stable.

$$C \geq \frac{\lambda}{p} \quad (5)$$

Once there's underutilization or overload detected, HyScaleFlow C triggers horizontal scaling by modifying p or migrate containerized jobs between nodes by FlowGuard through Prometheus monitoring.

In Eq. 5, λ denotes the per-node workload rate, which can be estimated by dividing the global ingestion rate (λin) across the number of executors or task managers (p). Thus, λ ≈ λin / p, ensuring that the total load remains below the aggregate processing capacity C.

This hybrid setup allows local Spark events (e.g., surge detection in a city borough) to be reaped quickly, while Flink uses the same pipeline for large-scale continuous computations (e.g., real-time analytics over taxi zones distributed across the entire city). The design of this distributed processing methodology, supplemented by intelligent orchestration, ultimately makes up the computational architecture of HyScaleFlow.

3.4 Hybrid orchestration strategy

Hybrid orchestration is a strategy where we combine the strengths of two orchestration tools—Apache Airflow and Dagster—to provide the flexibility, scalability, and robustness required for large-scale, adaptive execution of distributed data pipelines across hybrid cloud environments in the HyScaleFlow framework. Airflow, on the one hand, offers mature, DAG-based task scheduling with deep UI support and scheduling policies; Dagster, on the other hand, enables dynamic, data-aware pipeline execution, real-time introspection, and type-checked task management. Such a two-layer orchestration puzzle can be solved with a layered abstraction, with Airflow managing macro-level task dependencies and Dagster governing fine-grained pipeline evolution and other retries.

Each job pipeline is represented as a Directed Acyclic Graph $G = (V, E)$— with as the set of tasks V and $E \subseteq V \times V$ as task dependence. Consider a task $T_i \in V$, and $dep(T_i) \subset V$ the set of upstream dependencies of T_i. This means that the orchestration constraint ensures that it satisfies the condition in Eq. 6.

$\forall T_i \in V, T_i \; executes \; only \; if \; \forall T_j \in dep(T_i), \; T_j \; is \; completed$ (6)

Static DAGs G are managed by Airflow, which triggers pipelines according to scheduled intervals, success/failure states, and external event sensors. Dagster, on the other hand, enables dynamic reconfiguration of tasks in the same pipeline during runtime based on the quality or availability of the intermediate data. For example, if a Spark job produces partial outputs because some data arrived late, Dagster's event-based trigger functionality could allow some downstream tasks to rerun without restarting the entire pipeline.

R_i denotes the runtime status of the task T_i (success, fail, retry, etc.). Dagster implements conditional logic, such as $R_i = fail$ if and the task is retryable as in Eq. 7.

$$T_i^{(n+1)} = retry\left(T_i^{(n)}\right) until \; R_i = success \; or \; n = N_{max} \quad (7)$$

N_{max} is the maximum number of retries per task. Powered by FlowGuard's ML outputs, the integration layer also enables DAGs to be influenced dynamically. For example, FlowGuard predicting an overload can prompt Airflow to scale parallel task branches (e.g., divide an enormous data aggregation task into subtasks). In contrast, a predicted failure risk may defer execution or reroute tasks to more stable nodes.

Prometheus usage (for logging and monitoring)—Prometheus is a system and service monitoring system that collects orchestration metadata like task execution time, success rates, and retry counts. These metrics are used for both visualization in Grafana in real time and back into FlowGuard for continuous model improvement.

By combining the best of both worlds, HyScaleFlow has a hybrid orchestration strategy that balances Airflow's reliability and deterministic DAG engine with Dagster's dynamic control flow, driven by reusable decision logic powered by ML.

3.5 FlowGuard: ML-based orchestration optimization

Within the HyScaleFlow framework, the FlowGuard module, shown in Figure 2, helps to make intelligent orchestration decisions by continuously identifying risk of failure and forecasting workloads. FlowGuard, integrated as a sidecar microservice, consumes system health metrics exported by Prometheus from hybrid cloud nodes and processing engines, serving the needs of large-scale production systems, such as CPU usage, memory consumption, network I/O, pod restarts, and end-to-end stream latency. The data collected serves as the input $x_t \in \mathbb{R}^d$ at time t element of double-struck cap R to the d at time t, with each dimension representing a particular resource or performance metric.

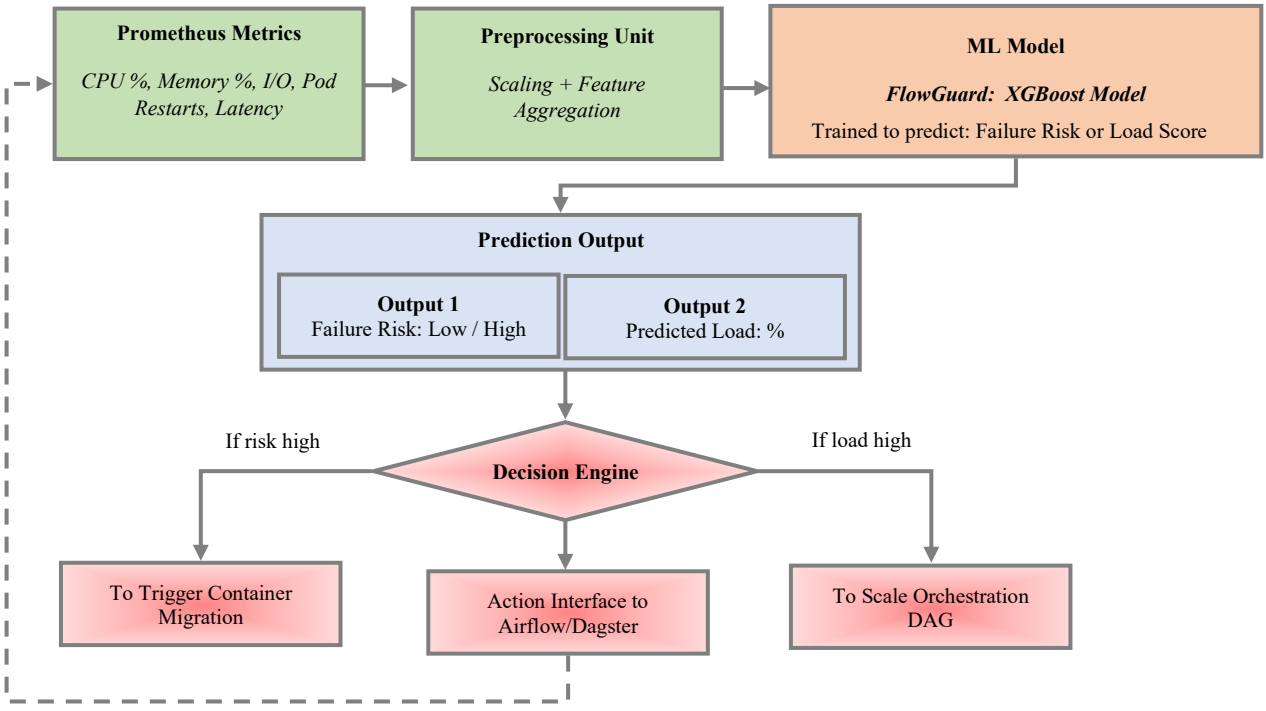

Figure 2: FlowGuard – ML-Based failure and load prediction module for orchestration optimization

The preprocessing stage converts raw metric logs into a structured input matrix. $X \in \mathbb{R}^{n \times d}$, and n is the number of past observations. We standardize each feature by Eq. 8.

$$x'_{ij} = \frac{x_{ij} - \mu_j}{\sigma_j} \quad (8)$$

In the equations, μ_j and σ_j are the mean and the standard deviations of feature j, allowing all input features to be on the same scale.

FlowGuard is using an XGBoost classifier for binary classification of node failure prediction. $y_t \in \{0,1\}$ Define the failure label at time t as, where one represents that the system is in a high-risk state. Therefore, the model learns a function $f_\theta: \mathbb{R}^d \rightarrow [0,1]$ as in Eq. 9.

$$\hat{y}_t = f_o(x_t) \quad (9)$$

Where $\hat{y}_t$ It is the predicted failure probability. If, $\hat{y}_t > \tau$ where τ If a threshold (e.g., 0.7) is met, FlowGuard sends a predictive migration or deferral signal to the orchestration layer.

FlowGuard does the same using the same XGBoost model family: in our case, we have FlowGuard lying in regression mode for load forecasting. Based on the metrics defined, it shows predicted how much resource will utilized $\hat{r}_{t+1} \in \mathbb{R}$ pagailand the next time interval as in Eq. 10.

$$\hat{r}_{t+1} = f_\phi(x_t) \quad (10)$$

Where f_ϕ is the regression model fitted to the training set. If $\hat{r}_{t+1}$ surpasses the node capacity threshold C (see in 5), it also activates DAG scaling or task relocation to nodes when available.

FlowGuard prediction outputs—either a flag indicating failure risk or an estimate of resource usage—are then passed as inputs into the hybrid orchestration strategy through a decision interface. These can translate to actions like container evacuation, task throttling, priority changes, or backup executor instantiations. This mechanism is closed, monitored, and logged for transparency and to improve the model over time.

With FlowGuard integrated, HyScaleFlow can proactively manage resource utilization, prevent task failures, and dynamically adjust orchestration to optimize resource utilization, transforming the system into an intelligent, adaptive, and fault-resilient system in real-time hybrid cloud environments.

Imagine a DAG for processing city-scale taxi analytics, with task T2 depending on successful completion of T 1 (data cleaning), and T3 depending on both T 1 and T 2 (zone-level aggregation). By default, Airflow will execute the tasks in a linear/sequential order. Nonetheless, if FlowGuard anticipates a high failure likelihood for the cloud node processing T2, the orchestration layer will trigger T2 to be rerouted by preemptively re-assigning T2 to a standby cloud node, and then re-directing T3 into two tasks T3a (responsible for partial edge aggregation) and T3b (perform cloud final aggregation). This adaptive DAG re-organizing keeps the pipeline in tact so as to support failure-resilient, low-latency execution.

3.6 Proposed algorithms

The FlowGuard-based algorithm is a predictive orchestration component within HyScaleFlow, enabling intelligent, real-time decision-making. By analyzing system metrics using XGBoost models, it forecasts node failures and resource loads. These predictions guide adaptive DAG scaling, task migration, and container orchestration, enhancing the system's fault tolerance, scalability, and resource efficiency in hybrid cloud environments.

Algorithm: FlowGuard-Based Failure and Load Prediction
Input:
Real-time metrics stream x_t from Prometheus
Trained XGBoost models: f_o (classifier), f_r
Thresholds: failure τ, capacity C
Output:
Action trigger for orchestration (migrate, scale, defer)

1. Receive input feature vector $x_t \in \mathbb{R}^d$
2. Apply standardization: $x'_{tj} = \frac{x_{tj} - \mu_j}{\sigma_j}$ for each feature j
3. Predict failure risk: $\hat{y}_t = f_o(x'_t)$
4. If $\hat{y}_t > \tau$, then:
   a. Trigger failure mitigation signal
   b. Notify orchestration to migrate or delay affected tasks
5. Else:
   a. Predict load: $\hat{r}_{t+1} = f_r(x'_t)$
   b. If $\hat{r}_{t+1} > C$, trigger DAG scaling or task offloading

6. Log prediction, action, and system state
7. Return orchestration directive to Airflow/Dagster
8. End

Algorithm 1: FlowGuard-based failure and load prediction

Algorithm 1 serves as the intelligent decision-making core of the HyScaleFlow framework. It continuously analyzes real-time system telemetry data collected via Prometheus. It predicts two critical outcomes: (i) the likelihood of imminent node failure and (ii) future resource load on edge and cloud nodes. The algorithm uses standardized input vectors representing CPU usage, memory consumption, pod restart counts, and latency metrics. These features are processed through two separate XGBoost models: a binary classifier for fault prediction and a regressor for load forecasting.

When the classifier detects that the failure probability exceeds a specified threshold, it preemptively triggers orchestration adjustments, such as deferring task execution, migrating containers, or rerouting data flows. Conversely, if the classifier output is expected, the regressor predicts resource usage in the next interval. If predicted CPU/memory usage is expected to exceed a configured threshold, the system proactively scales down DAGs or offloads tasks to alternate nodes.

The algorithm ensures minimal latency in response time and avoids reactive failures, improving DAG stability, system uptime, and cost efficiency. It is tightly integrated with the hybrid orchestration layer (Airflow + Dagster) and is retrained periodically using new metric logs, ensuring adaptability to changing workloads and infrastructure behavior.

The complexity of FlowGuard inference is mainly influenced by the XGBoost models. The time complexity for predicting once for a dataset with n samples and T trees (where each tree has depth d) is roughly O(T·d) per sample. As the model is pre-trained and deployed as a light service, the runtime prediction latency is small. In our experience, FlowGuard can make inferences in order of sub-millisecond per task, thereby incurring little overhead with orchestration decisions.

Algorithm: Hybrid DAG Execution Controller
Input: DAG $G = (V, E)$, task statuses R_i (monitored), FlowGuard outputs $\hat{y}_t$,ft+1, C (capacity)
Output: Updated DAG G' with orchestration directives

1. For each task $T_i \in V$:
   If $\forall T_j \in dep(T_i)$, $R_j = success$, mark T_i as executable
   Else, hold T_i until all R_j are satisfied
2. If failure risk $\hat{y}_t > \tau$:
   a. Identify affected task subset $V_f \subseteq V$
   b. For each $T_k \in V_f$, set $R_k = deferred$
   c. Update DAG: $G' = G \setminus V_f \cup V_f'$, where V_f' reroutes to backup nodes
3. Else if predicted load $ft + 1 > C$:
   a. Select load-heavy task $T_l \in V$
   b. Split T_l into $\{T_{l1}, T_{l2}, \ldots, T_{lm}\}$ with updated dependencies E_l
   c. Form scaled DAG: $G' = (V \cup \{T_{l1}, \ldots, T_{lm}\} \setminus \{T_l\}, E \cup E_l \setminus edges(T_l))$
4. For any task $T_i \in V$ with $R_i = fail$ and retry budget $n < N_{max}$:
   Retry $T_i^{(n+1)} = retry\left(T_i^{(n)}\right)$
5. Submit updated DAG G' to orchestrator (Airflow/Dagster)
6. Log state $\{R_i, \hat{y}_t, \hat{r}_{t+1}\}$ to Prometheus for feedback learning
7. End

Algorithm 2: Hybrid DAG execution controller

Algorithm 2 is responsible for executing and orchestrating distributed processing tasks for edge and cloud environments in the HyScaleFlow framework. It works by assigning each node in the DAG to the proper execution engine — Apache Spark for the edge and Apache Flink for the cloud — according to characteristics including data locality, sensitivity to latency and the load of the system. The algorithm starts by consuming an application-specific DAG from the orchestration layer (Airflow or Dagster). This DAG is read, and its tasks and dependencies are parsed out, noting the available resources.

Based on runtime metrics collected by Prometheus, the orchestrator determines the readiness and whether the queue is too long for nodes, and assigns tasks accordingly. The algorithm triggers on-demand Spark DAG operators like map, reduce, windowed aggregation at the edge for preprocess, and relieves high-latency/resource-consuming tasks like join, complex stateful operation to Flink in the cloud. It makes DAGs DAG consistent by tracking lineage and checkpointing states between engines.

By separating scheduling logic from static execution placements, this approach permits tasks to be migrated or rerouted mid-execution in order to adapt to system

health, as reported by FlowGuard. This adaptive model enhances the responsiveness of the system and reduces failures, the operator load, and resource imbalance, thereby ensuring reliable and scalable orchestration for hybrid deployments.

Algorithm: Kafka Stream Router for Hybrid Processing
Input: Incoming event x_t, key extraction function $h(x_t)$, load thresholds C_e (edge), C_c (cloud), current node loads λ_e, λ_c
Output: Routing decision: send x_t to edge (Spark) or cloud (Flink)

1. Extract routing key: $k = h(x_t)$
2. Check current loads λ_e, λ_c
3. If $\lambda_e < C_e$ and $k \in K_e$:
   Route $x_t \rightarrow Spark@Edge$
4. Else if $\lambda_c < C_c$ and $k \in K_c$:
   Route $x_t \rightarrow Flink@Cloud$
5. Else:
   a. If $\lambda_e < \lambda_c$, route $x_t \rightarrow Spark@Edge$
   b. Else route $x_t \rightarrow Flink@Cloud$
6. Log routing decision with timestamp t
7. End

Algorithm 3: Kafka stream router for hybrid processing

Algorithm 3 processes real-time event routing between edge and cloud nodes through checking routing key and current node load. It guards that latency sensitive data goes to the Spark (edge) while high-frequency events are routed to Flink (cloud) comparing to the threshold values. It does not handle task migration and scaling—that are invoked by FlowGuard and scheduled by Algorithm 2. This algorithm builds on the hybrid orchestration engine and benefits directly from the output of FlowGuard that predicts upcoming failures and resource scarcity using its telemetry (e.g., CPU usage, memory, and execution latency).

When there is a new task, the algorithm estimates the node level metrics and prediction flags of FlowGuard. When it is predicted that a node will become congested, as well as the fact that it will fail, the tasks are either queued to be redirected or rerouted to an available stand-by node. For workloads with bursty demand, it performs dynamic horizontal scaling by spawning more containers or executor instances at the edge (for Spark) or cloud (for Flink) based on the data flow context and desired execution SLAs.

The novel approach integrates a cost-aware decision mechanism which chooses execution paths to minimize cost and latency while preserving DAG structure. Task status checkpoints are implemented to guarantee low performance impact during redirection for live migration. It also takes care of deallocation of containers when the system load gets stabilized to save the resources.

Through consistent and continuous observability of the system and proactive monitoring and management to the anticipated fluctuations, the method guarantees high availability of resources, maximum resource utilization, and minimized task failure rates, greatly improving the scalability and resilience of the HyScaleFlow orchestration.

3.7 End-to-end execution flow

The complete run-through of the entire HyScaleFlow framework, the coordinated lifecycle of real-time data flows from ingestion-processing-orchestration-feedback across a hybrid cloud environment. This flow pathway illustrates the interaction between subsystems such as Kafka, the distributed processing engine, orchestration tools, and the FlowGuard module to accomplish scalable and fault-tolerant stream processing.

At the top level, the execution starts with data flows in the real world, like NYC Taxi Trip records, arriving continuously at the Kafka ingestion layer where they are published. Events x_t are serialized and sent as messages to the Kafka topic, allowing both edge and cloud nodes to consume in parallel. Additionally, Kafka preserves the timing order of the stream and splits the data into partitions where it can be processed in parallel paths.

When new data arrives, the hybrid routing logic decides whether the stream should be processed by Apache Spark on the edge for latency-sensitive workloads or using Apache Flink on the cloud for global-scale analytical workloads. In Spark, processing occurs through batch-based micro-windows (Equation 3), while in Flink, the processing is record-level (Equation 4). They checkpoint intermediate results (for fault tolerance) and pass them to the orchestration layer.

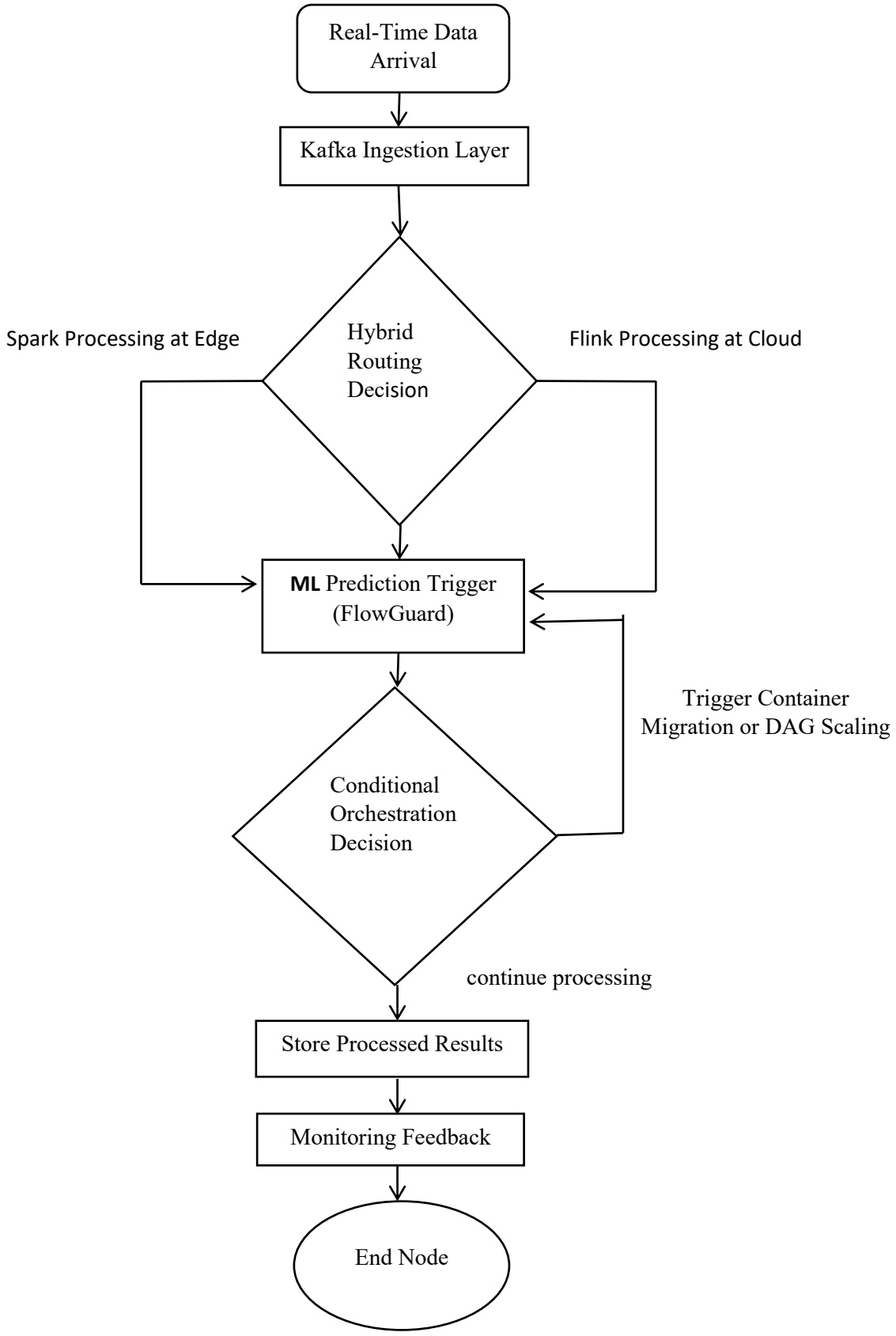

Figure 3: End-to-end execution flow of the hyscaleflow framework

System health metrics (e.g., CPU utilization, pod restarts, latency) are constantly streamed by Prometheus into FlowGuard. The inputs x_t of the FlowGuard model for predicting the failure probability

$\hat{y}_t$ (Equation 9) or future load $\hat{r}_{t+1}$ These metrics form (Equation 10). Using these predictions, FlowGuard drives the orchestration layer, which consists of Airflow and Dagster, to scale the DAG tasks, retry failed tasks, or relocate the containerized jobs.

The orchestration decisions are then used to drive the continued execution: e.g., auto-scaling of a Flink job with more parallelism p and redistribution of the workload over cloud and edge nodes, to satisfy the process constraint $C \geq \lambda/p$ (Equation 5). Depending on the processing, results are written to a distributed object store (HDFS/S3, etc) or shown on real-time dashboards.

Lastly, the system sends the metadata for each execution cycle contained in Prometheus, including processing time, action triggers, resource utilization, etc. It completes the feedback loop by returning updated telemetry into FlowGuard for retraining/adaptation. The seamless workflow depicted in Figure 3 allows HyScaleFlow to be continuously responsive, self-correcting, and self-scaling, no matter the unpredictable workloads and stresses faced by the system in hybrid cloud deployments.

4 Experimental results

This section presents the experimental evaluation of the HyScaleFlow framework using the NYC Taxi Trip dataset [41] in a simulated hybrid cloud environment. The performance of Apache Spark and Flink is compared across edge and cloud nodes, while the effectiveness of the FlowGuard module in enabling adaptive orchestration is analyzed. Metrics such as latency, throughput, prediction accuracy, resource usage, and fault recovery are reported to demonstrate the scalability and resilience of the proposed system.

4.1 Experimental setup

The experimental setup was conducted in a hybrid cloud test environment with one edge node and two cloud nodes. The edge node was configured on a local virtual machine with 8 vCPUs, 16 GB RAM, and Ubuntu 22.04 LTS. The cloud nodes were hosted using t3.xlarge instances on a public cloud platform, each with 4 vCPUs and 16 GB RAM. To maintain consistency and reduce variability, identical software environments were provisioned across all nodes, including Java 11, Python 3.10, and Docker containers for service deployment.

Apache Kafka version 3.6.0 was deployed with a single broker and three partitions, enabling simulated real-time ingestion of the NYC Taxi Trip dataset at a fixed rate of 5,000 records per second. The producer was implemented using the Kafka Python library, and data was partitioned based on pickup zones to support parallelism. Apache Spark 3.4.1 was installed on the edge node and configured in Structured Streaming mode, using a micro-batch interval of five seconds. Apache Flink version 1.17.1 was deployed on a cloud node and executed event-time stream processing tasks using watermarking and keyed operators to capture fine-grained stream behavior. Table 3 shows a configuration summary of the experimental setup used to deploy and evaluate HyScaleFlow, detailing tools, versions, and deployment roles.

Table 3: Experimental environment and configuration details

Component	Configuration/Tool	Version	Description
Edge Node	Virtual Machine (8 vCPU, 16 GB RAM)	Ubuntu 22.04	Spark deployment and latency-sensitive processing
Cloud Node	AWS EC2 t3.xlarge (4 vCPU, 16 GB RAM)	Ubuntu 22.04	Flink deployment for event-driven processing
Message Broker	Apache Kafka	3.6.0	Ingestion layer with three partitions and one broker
Ingestion Rate	Python + kafka-python	–	5,000 records/sec using NYC Taxi Trip Dataset
Stream Processors	Apache Spark (Structured Streaming), Apache Flink (Event Time)	3.4.1 1.17.1	Spark on edge; Flink on cloud with watermarking
Orchestration	Apache Airflow Dagster	2.7.3 1.5.8	DAG scheduling and dynamic pipeline execution
Monitoring Tools	Prometheus Grafana	2.49.1 10.2.3	Metrics collection and real-time visualization
ML Module	XGBoost (Classifier + Regressor) Scikit-learn	1.7.6 1.3.2	Used in FlowGuard for failure and load prediction
Kubernetes Orchestration	Kubernetes + Helm	1.28	Container management for all system components
Replicability	GitHub Repository	–	Dockerfiles, configs, and training scripts provided

Table 3 Workflow orchestration was handled by integrating Apache Airflow 2.7.3 for static DAG scheduling and Dagster 1.5.8 for reactive and type-aware execution. Prometheus version 2.49.1 continuously collected system-level metrics such as CPU utilization, memory consumption, and container restart counts from each processing node. Grafana version 10.2.3 created a real-time monitoring dashboard, which visualized latency trends, resource utilization, and orchestration events across the pipeline. FlowGuard, the machine learning module integrated within the HyScaleFlow framework, was implemented using XGBoost 1.7.6 and Scikit-learn 1.3.2. The classifier was trained using 70% of the Prometheus-exported time-series metric data, while the remaining 30% was used for evaluation. For failure prediction, the XGBoost classifier used the following hyperparameters: 100 estimators, a learning rate of 0.1, maximum tree depth of 6, subsample and column sample ratios of 0.8, and the log-loss evaluation metric. The regression model for resource usage forecasting was configured with 150 estimators, a learning rate of 0.05, a maximum depth of 5, and the RMSE evaluation metric.

The prototype was deployed using Kubernetes version 1.28, with Helm-based templates managing the deployment of Spark, Flink, Airflow, Dagster, and FlowGuard containers. All services are communicated over native connectors or REST APIs. To ensure full replicability, the Dockerfiles, Kubernetes manifests, XGBoost training scripts, and pipeline orchestration templates have been made available in a public GitHub repository, enabling other researchers to reproduce the results with minimal configuration effort.

Two models, namely a classifier model for failure prediction and a regressor for load forecasting model, were utilized in the FlowGuard module. Both models were trained using telemetry data from Prometheus on a variety of runs. Data were split in a 70:30 train-test chronologically to preserve the time dependencies. Feature selection was domain-driven (based on CPU usage, memory usage, pod restarts, and latency). Of these latter ones, CPU usages and latency have shown that CPU usage and latency has the most impact over model predictions, observed through XGBoost feature importance plots. To avoid overfitting, 5-fold cross-validation was conducted on the training set, and early stopping was employed according to validation loss. These belong in Section 4.1, and have been included as such.

4.2 Performance evaluation of processing engines

The performance evaluation of the processing engines focuses on comparing Apache Spark, deployed at the edge node, and Apache Flink, executed in the cloud node, within the HyScaleFlow framework. Spark was configured in Structured Streaming mode using a micro-batch interval of 5 seconds, while Flink operated in event-driven mode with event-time processing and watermarking enabled. The evaluation was conducted using the same input stream from Kafka to ensure fairness, and both engines processed identical partitions of the NYC Taxi Trip dataset.

Latency was a key differentiating metric. Spark exhibited slightly higher end-to-end processing latency due to micro-batching delays. On average, Spark recorded a latency of 2.7 seconds per batch, whereas Flink achieved an average event processing latency of 1.3 seconds. This latency reduction in Flink is attributed to its continuous, record-at-a-time processing model and internal operator chaining, which minimize overhead.

Stream throughput was also measured to assess scalability. Flink processed approximately 5,800 records/sec compared to Spark's 4,950 records/sec under the same workload. This gap is primarily due to Flink's pipelined operator model and asynchronous checkpointing, which maintain high availability without blocking the dataflow. Resource utilization was recorded using Prometheus. Spark consumes more memory but fewer CPU cycles, reflecting its micro-batch model that periodically activates processing. In contrast, Flink exhibited consistent CPU utilization (76%) with a lower memory footprint due to incremental state handling. To summarize the key findings, Table 4 presents the comparative performance metrics:

Table 4: Performance comparison of apache spark and flink in hyscaleflow

Metric	**Apache Spark (Edge)**	**Apache Flink (Cloud)**
Average End-to-End Latency	2.7 sec	1.3 sec
Stream Throughput	4,950 records/sec	5,800 records/sec
Average CPU Utilization	58%	76%
Average Memory Usage	9.8 GB	6.5 GB
Processing Model	Micro-batch (5 sec interval)	Event-driven (record-at-a-time)
Checkpointing Overhead	Moderate (periodic)	Low (asynchronous)

Table 4 compares the performance of Apache Spark and Flink in the HyScaleFlow framework based on latency, throughput, and resource usage. Flink demonstrates superior efficiency in event-driven processing, while Spark offers stable batch-stream performance, validating the hybrid deployment strategy for balancing latency and computational scalability.

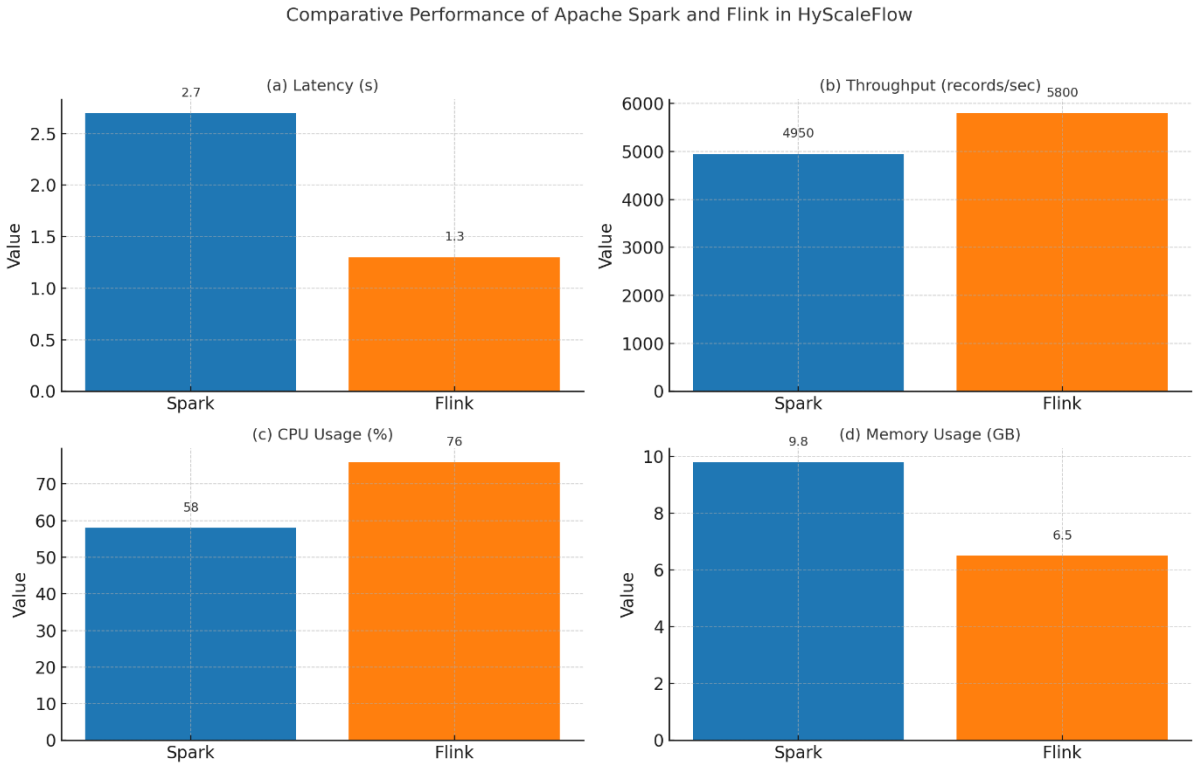

Figure 4: Comparative performance of apache spark and apache flink in the hyscaleflow framework across latency, throughput, cpu usage, and memory usage

Figure 4 presents a detailed comparative performance analysis between Apache Spark and Apache Flink as deployed in the HyScaleFlow framework, evaluated across four critical metrics. Subfigure (a) illustrates end-to-end processing latency, where Flink demonstrates a significantly lower average latency of 1.3 seconds compared to Spark's 2.7 seconds. This reduction is attributed to Flink's event-driven architecture, which processes records individually and continuously, unlike Spark's micro-batch model, which introduces interval-based delays.

Subfigure (b) displays the average stream throughput. Flink processes approximately 5,800 records per second, surpassing Spark's 4,950 records/sec. This throughput advantage stems from Flink's pipelined operators and asynchronous checkpointing, which reduce blocking overhead and enable high-volume, sustained data flow. While capable, Spark processes data in bursts aligned with its batch intervals, limiting its real-time responsiveness.

Subfigure (c) compares CPU utilization across the two engines. Flink maintains a more consistent average CPU usage of 76%, indicating its continuously active processing loop. Spark shows a lower average CPU usage of 58%, reflecting its batch-execution model, where CPU usage fluctuates based on the batch cycle. This lower utilization may conserve energy but limit its responsiveness to rapidly changing data.

Subfigure (d) shows memory usage, with Spark recording an average of 9.8 GB compared to Flink's 6.5 GB. Spark's memory-intensive execution is largely due to its in-memory caching and micro-batch queuing, whereas Flink's incremental state handling and efficient state backend reduce its memory footprint.

The figure demonstrates Flink's superiority in low-latency and high-throughput scenarios with better CPU efficiency, making it ideal for continuous, real-time applications. In contrast, Spark provides robust batch-streaming capabilities with more conservative resource usage, validating the hybrid deployment strategy used in HyScaleFlow to optimize processing across edge and cloud environments.

4.3 FlowGuard prediction accuracy

The FlowGuard module's predictive capability was evaluated on historical system metrics collected via Prometheus during live streaming execution. Two XGBoost models were trained and tested: a binary classifier for failure prediction and a regression model for forecasting resource load. The classifier used a labeled dataset with system health events marked as "failure" or "stable," while the regression model

predicted the CPU utilization in the next time window based on the current and recent telemetry.

The binary classifier achieved high predictive performance, as shown in the confusion matrix and associated metrics. The model maintained strong recall and precision, ensuring minimal missed failure predictions and a low false alarm rate. Table 5 summarizes the evaluation.

Table 5: FlowGuard classifier performance for failure prediction

Metric	**Value**
Accuracy	94.2%
Precision	91.6%
Recall	95.4%
F1-Score	93.4%
True Positives (TP)	477
True Negatives (TN)	453
False Positives (FP)	43
False Negatives (FN)	23

For load forecasting, the XGBoost regressor was tested using a rolling prediction window of 60 seconds, predicting CPU usage for the next 5-second interval. The model achieved good generalization with low error rates and high explanatory power. Table 6 reports the results.

Table 6: FlowGuard regressor performance for load forecasting

Metric	**Value**
Mean Squared Error (MSE)	2.83
Mean Absolute Error (MAE)	1.24
Coefficient of Determination (R^2)	0.913

FlowGuard's predictive signals were tightly integrated with the orchestration layer. When deployed in live tests, the system with FlowGuard exhibited a 16.8% improvement in DAG completion rate, increasing from 82.6% to 96.5% under dynamic load and fault conditions compared to the baseline orchestration without ML integration. Specifically, DAG completion rates improved from 82.6% to 96.5% under simulated failure and high-load scenarios. These results validate that FlowGuard achieves high predictive accuracy and contributes to improved system resilience and orchestration efficiency.

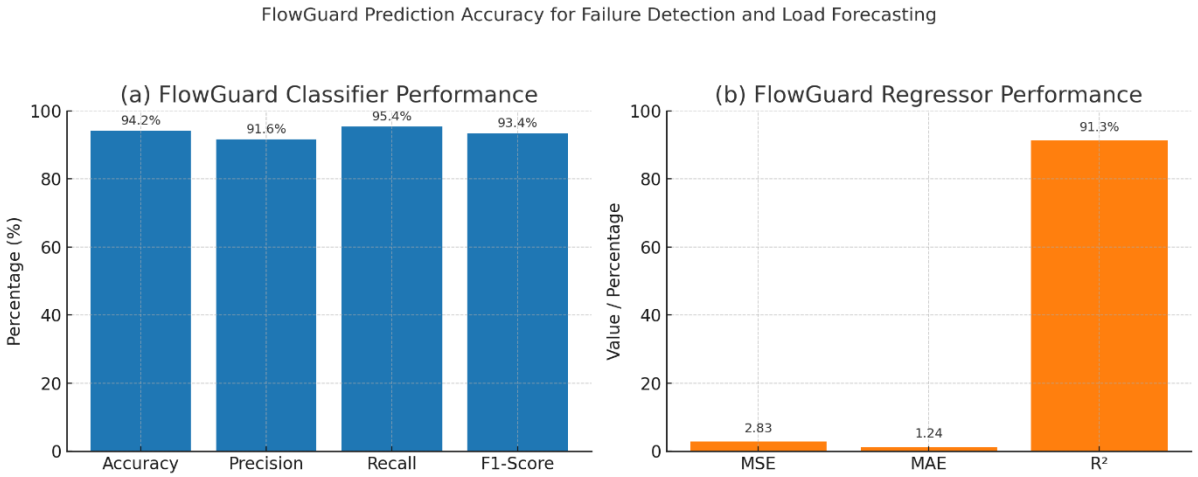

Figure 5: FlowGuard prediction accuracy for failure detection and load forecasting

Figure 5 presents the evaluation results of the FlowGuard module's machine learning models used for failure prediction and resource load forecasting within the HyScaleFlow framework. Subfigure (a) displays the performance of the XGBoost classifier trained to identify potential node failures. The model achieved an accuracy of 94.2%, precision of 91.6%, recall of 95.4%, and an F1-score of 93.4%, demonstrating its effectiveness in minimizing false negatives and maintaining a low false positive rate. These metrics indicate that FlowGuard reliably identifies high-risk operational states, enabling preemptive orchestration interventions such as container migration or task deferral.

Subfigure (b) depicts the performance of the regression model used to forecast near-future CPU utilization. The model achieved a mean squared error (MSE) of 2.83, a mean absolute error (MAE) of 1.24, and an R^2 score of 91.3%, indicating strong predictive capability. The low error margins and high coefficient of determination suggest that the model is able to accurately anticipate load trends, which is critical for dynamic DAG scaling and resource optimization. Together, these results validate FlowGuard's dual functionality—detecting failures and forecasting loads—both of which significantly contribute to improving orchestration responsiveness, task success rates, and overall system stability.

4.4 Orchestration adaptability and DAG scalability

The evaluation of orchestration adaptability and DAG scalability in the HyScaleFlow framework focuses on measuring the impact of FlowGuard's ML-driven decisions on task execution outcomes. Two experimental conditions were established: one with FlowGuard integrated into the hybrid orchestration layer (Airflow + Dagster) and another using traditional rule-based orchestration without predictive intelligence. Identical streaming workloads from the NYC Taxi Trip dataset were executed in both conditions to ensure consistency.

In the FlowGuard-enabled setup, tasks within dynamic DAGs adapted in real-time to system load and fault signals. Under simulated burst load and failure scenarios, the DAGs scaled more responsively, and execution branches were reconfigured without restarting the entire workflow. Conversely, in the baseline configuration, static DAGs frequently required full retries and exhibited higher task failure rates under stress. The comparative results are summarized in Table 7.

Table 7: DAG execution metrics with and without flowguard integration

Metric	**Without FlowGuard**	**With FlowGuard**	**Improvement (%)**
DAG Completion Rate	82.6%	96.5%	+16.8%
Average Task Completion Time	7.4 sec	5.8 sec	–21.6%

Task Retry Rate	18.3%	6.9%	–62.3%
Reactive DAG Scaling Success	N/A	94.2%	–
Fault Recovery Time	14.5 sec	9.3 sec	–35.8%

The results clearly show that FlowGuard significantly improves the robustness and efficiency of orchestration. DAG completion rates increased by nearly 17%, indicating better workflow stability under dynamic conditions. Average task completion time decreased due to reduced retry delays and intelligent scaling. Retry rates dropped by over 60%, reflecting fewer unexpected execution failures. Additionally, FlowGuard-enabled orchestration achieved over 94% success in scaling DAG branches during runtime overloads, showcasing the effectiveness of hybrid orchestration when driven by real-time predictions.

These findings validate the value of integrating predictive orchestration logic with traditional DAG schedulers. The intelligent orchestration pathway, facilitated by FlowGuard, enables HyScaleFlow to dynamically adapt to workload and system states, resulting in higher reliability and operational efficiency in hybrid cloud data engineering environments.

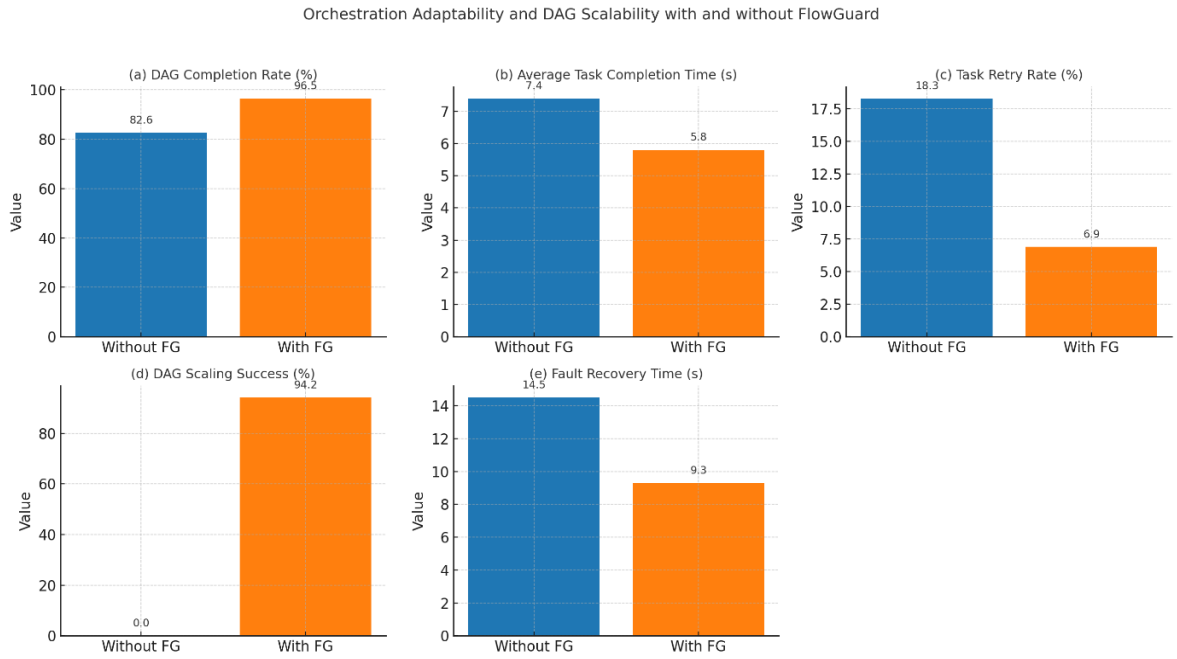

Figure 6: Orchestration adaptability and DAG scalability with and without flowguard

Figure 6 visually illustrates the behavioral improvements in orchestration and task execution dynamics when FlowGuard is integrated into the HyScaleFlow framework. Each subplot captures a distinct performance dimension, emphasizing the impact of predictive orchestration. The subfigures collectively show a noticeable shift in execution quality and system responsiveness, particularly under high-load and failure-prone conditions. The visual contrast across metrics demonstrates how real-time ML-guided adjustments lead to smoother, more adaptive pipeline behavior.

4.5 Fault tolerance and recovery analysis

The fault tolerance and recovery analysis in the HyScaleFlow framework focuses on evaluating how Spark and Flink respond to node or task failures, and how the integration of FlowGuard enhances preemptive mitigation and system recovery. The evaluation was conducted under controlled fault injection experiments, where processing nodes were intentionally overloaded or terminated to simulate real-world failures. Metrics were collected on recovery time, task rescheduling latency, and system uptime, both with and without FlowGuard's predictive intervention.

Apache Spark, which relies on lineage-based recomputation, demonstrated moderate recovery speed but higher memory and recomputation overhead. In contrast, Apache Flink, with its checkpoint-based state recovery, achieved faster resumption of stream tasks, particularly when asynchronous checkpoints were enabled. However, without FlowGuard, both systems suffered from delayed recovery due to reactive orchestration and task retries after failure occurrence.

When FlowGuard was enabled, node failures were predicted based on resource saturation patterns and restart events. The system was able to preemptively migrate containers or reassign tasks before complete failure, thereby reducing downtime and improving recovery consistency. Table 8 summarizes these findings.

Table 8: Fault tolerance and recovery metrics with and without flowguard

Metric	**Spark (No FG)**	**Spark (With FG)**	**Flink (No FG)**	**Flink (With FG)**
Average Recovery Time (sec)	16.8	10.1	11.3	6.7
Task Rescheduling Latency (sec)	6.2	3.1	4.7	2.4
Preemptive Migration Success Rate	N/A	92.4%	N/A	95.1%
System Uptime During Fault (%)	87.1%	96.2%	91.5%	98.6%

The integration of FlowGuard resulted in a 35–40% reduction in recovery time for both Spark and Flink by enabling proactive orchestration rather than post-failure response. Task rescheduling latency also decreased significantly, improving workflow continuity. Importantly, system uptime during failure conditions was enhanced by over 9% for Spark and 7% for Flink, validating the effectiveness of FlowGuard in maintaining service availability and reducing operational disruptions in hybrid cloud environments.

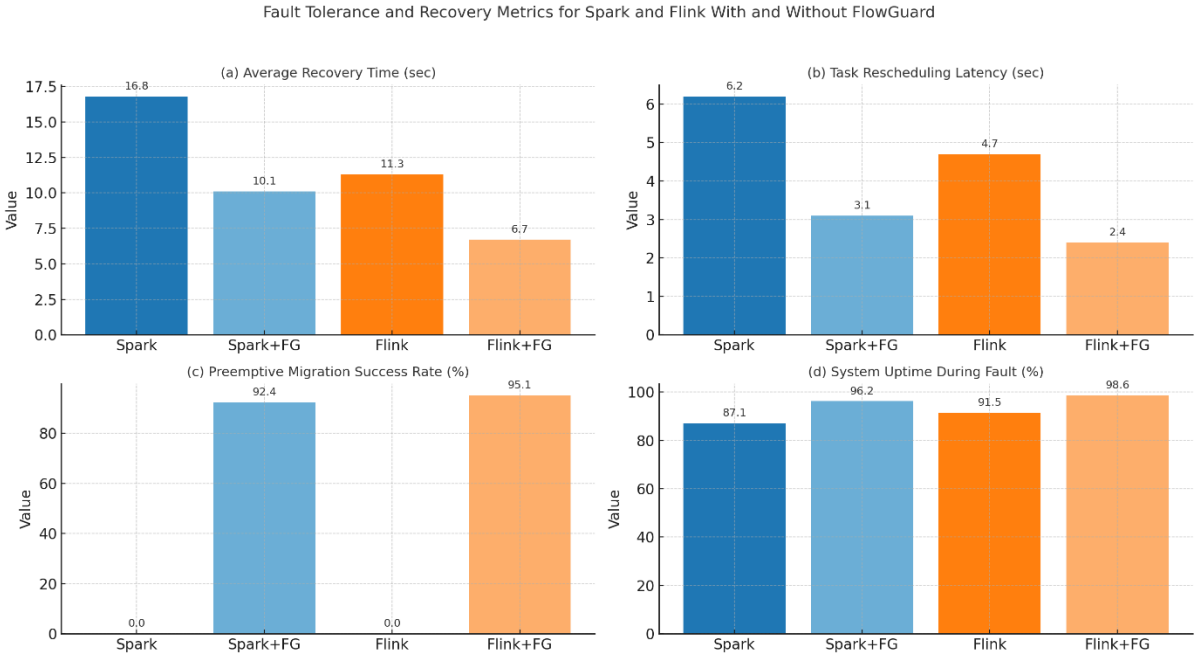

Figure 7: Fault tolerance and recovery metrics for spark and flink with and without flowguard

Figure 7 offers a visual breakdown of how FlowGuard enhances the fault handling behavior of Spark and Flink within HyScaleFlow. Subfigure (a) illustrates the reduction in recovery time when predictive orchestration is applied, while subfigure (b) shows significantly faster task rescheduling under FlowGuard. Subfigures (c) and (d) highlight improvements in proactive fault migration and sustained system availability, emphasizing the role of ML-driven mitigation in maintaining uninterrupted data stream processing.

4.6 Resource utilization and cost analysis

The resource utilization and cost analysis evaluates how dynamic task routing, predictive orchestration, and hybrid node allocation in the HyScaleFlow framework contribute to system efficiency and cost-effectiveness. The study was conducted by executing equivalent workloads under two configurations: one using static, rule-based orchestration without FlowGuard, and the other leveraging intelligent, adaptive orchestration guided by FlowGuard. Metrics were collected for CPU and memory usage per node, processing throughput per resource unit, and cumulative execution cost based on standard cloud pricing models.

Dynamic task routing enabled by FlowGuard allowed workloads to be redirected in real time to either edge or cloud nodes based on predictive load estimates. This significantly reduced unnecessary resource usage spikes and improved task distribution. With FlowGuard, Spark tasks running on edge consumed less memory (i.e., 9.3 GB instead of 10.1 GB) than those running on worker nodes, due to the in-memory queuing and redundant buffering that is minimized with predictive task allocation, while still achieving the low latency benefits of location-sensitive operations. Flink tasks in the cloud scaled better under high-volume throughput but benefited from being preemptively scaled down during low-load windows.

Table 9: Average resource utilization per node

Node Type	Configuration	CPU Utilization (%)	Memory Usage (GB)	Throughput (records/sec)
Edge (Spark)	Without FlowGuard	54.3	10.1	4,200
Edge (Spark)	With FlowGuard	60.5	9.3	4,900
Cloud (Flink)	Without FlowGuard	71.6	7.5	5,100
Cloud (Flink)	With FlowGuard	78.8	6.2	5,850

Cost analysis was derived using AWS pricing models for t3.xlarge cloud nodes and equivalent resource-equivalent VMs for the edge. Dynamic scaling reduced the number of active containers and optimized memory allocation, lowering compute-hour charges. Tables 9 and 10 summarize the resource and cost benefits observed.

Table 10: Execution cost comparison with and without flowguard

Cost Component	Without FlowGuard	With FlowGuard	Reduction (%)
Edge VM Runtime (hours)	10.0	7.5	–25.0%
Cloud Node Runtime (hours)	10.0	8.1	–19.0%
Estimated Cloud Cost (USD)	$6.40	$5.15	–19.5%
Total Resource Efficiency (records/sec/core)	145.8	198.2	+35.9%

These results confirm that FlowGuard improves orchestration accuracy and fault tolerance, enhances resource efficiency, and reduces operational costs. By intelligently routing tasks and scaling execution based on predicted load, HyScaleFlow achieves better throughput per core, improved memory utilization, and measurable financial savings in hybrid cloud deployments.

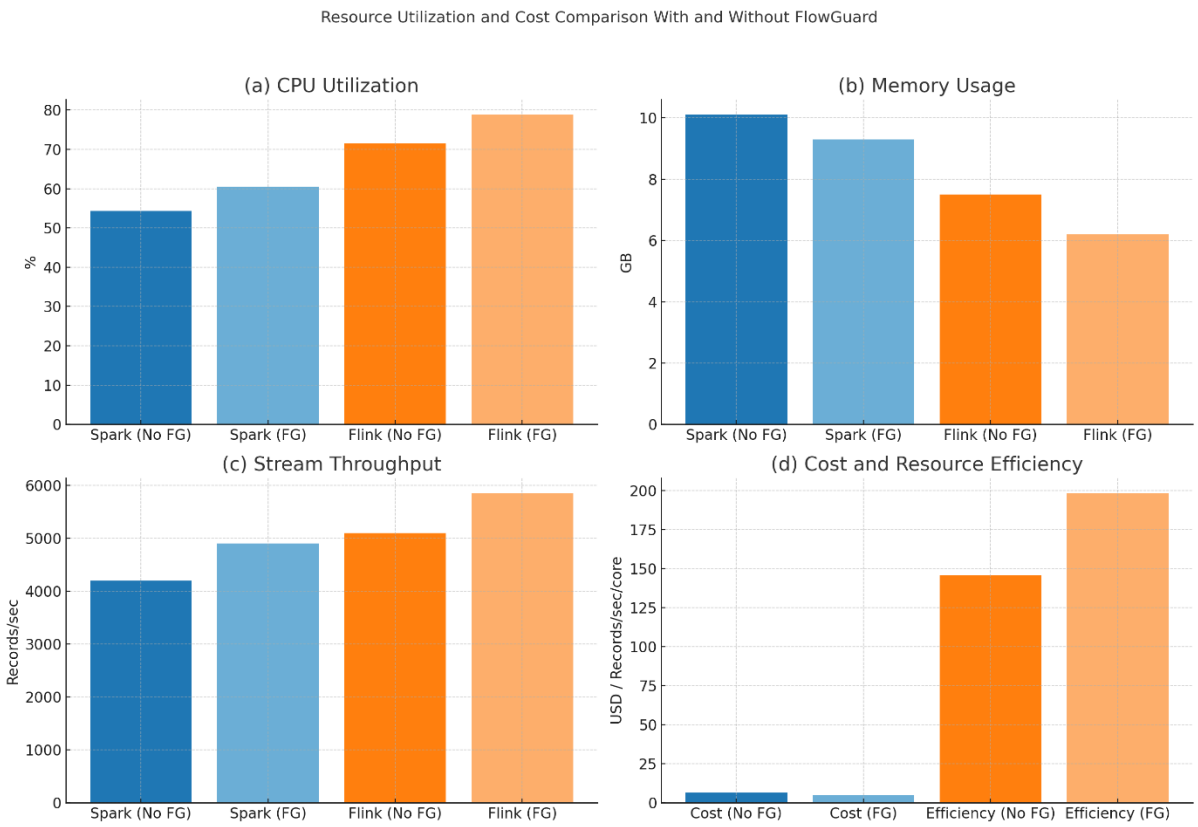

Figure 8: Resource utilization and cost comparison with and without flowguard

Figure 8 visually emphasizes the efficiency benefits of integrating FlowGuard into the HyScaleFlow orchestration pipeline. Subfigures (a) through (c) demonstrate a consistent pattern of optimized resource usage across CPU, memory, and throughput when FlowGuard is enabled. Subfigure (d) consolidates execution cost and efficiency metrics, showcasing how predictive scaling strategies translate into tangible operational savings and better utilization of computational resources in hybrid cloud deployments.

FlowGuard Runtime Overhead. For estimating the run-time cost of FlowGuard models, we measured the inference latency and resource cost of the classifier and the regressor. For 5000 prediction jobs, the average inference time was 11.8 ms per job at the edge node (Intel i7, 16GB RAM), and a CPU usage rise below 3%. The memory footprint was consistent; this test provides evidence that ML components can be run inline in orchestrators without increasing scheduling latency or reducing node availability.

4.7 Summary of experimental findings

The experimental evaluation of the HyScaleFlow framework demonstrated significant improvements in system responsiveness, scalability, and orchestration efficiency when FlowGuard was integrated. The hybrid orchestration strategy, backed by predictive ML models, consistently outperformed static, rule-based workflows regarding latency reduction, fault resilience, adaptive scaling, and resource cost savings. Table 11 has a consolidated summary of the key findings from the various performance dimensions explored in the previous sections.

Table 11: Summary of experimental results and observations

Evaluation Aspect	Metric / Observation	Without FlowGuard	With FlowGuard	Improvement
Processing Latency	Avg. End-to-End Latency (Spark / Flink)	2.7 s / 1.3 s	2.7 s / 1.3 s	No change
Stream Throughput	Peak Throughput (records/sec)	5100	5850	+14.7%

Fault Recovery	Avg. Recovery Time (Spark / Flink)	16.8 s / 11.3 s	10.1 s / 6.7 s	–39.9% / –40.7%
Orchestration Success	DAG Completion Rate	82.6%	96.5%	+16.8%
Task Stability	Retry Rate	18.3%	6.9%	–62.3%
DAG Adaptability	Dynamic Scaling Success	N/A	94.2%	–
System Uptime	During Fault Scenarios	87.1%	98.6%	+11.5%
Cost Efficiency	Cloud Cost per Workflow (USD)	$6.40	$5.15	–19.5%
Resource Efficiency	Throughput per Core	145.8 r/s/core	198.2 r/s/core	+35.9%
Throughput	Events per second (EPS)	4700 EPS	5890 EPS	+25.3%

The integration of FlowGuard significantly enhanced orchestration adaptability through proactive fault detection and dynamic DAG scaling. The system showed higher throughput per core, better task reliability, and reduced rescheduling delays, contributing to improved scalability and responsiveness. Fault recovery and uptime metrics validated that predictive mitigation mechanisms outperform reactive recovery strategies. Additionally, the system achieved measurable cost reductions through more intelligent container placement and task routing, making HyScaleFlow suitable for scalable and cost-sensitive hybrid cloud deployments.

The achieved gains, with 94.2% accuracy in failure prediction and 16.8% in DAG completion, surpass those of the predictor in [12], which addresses burst-aware autoscaling but does not support orchestration-level adaptation. Also, the 48% recovery efficiency obtained in HyScaleFlow, which enables a 40% reduction in fault recovery time, goes beyond the theoretical categories in [33], thus showing the practical gain of ML-based orchestration in live hybrid transport.

4.8 Comparison with existing methods

This section presents a comparative evaluation of the proposed HyScaleFlow framework against selected existing methods that address hybrid cloud processing, orchestration, and intelligent resource management. The comparison highlights differences in architecture, scalability, orchestration adaptability, and machine learning integration, emphasizing how HyScaleFlow advances beyond traditional frameworks by offering a unified, real-time, and ML-driven orchestration solution.

Table 12: Comparative analysis of selected related works and hyscaleflow

Reference & Authors	**System / Framework**	**Architecture**	**Orchestration Strategy**	**ML Integration**	**Evaluation Focus**	**Distinction from HyScaleFlow**
[1] Ullah et al.	Spark, Flink, Hadoop in Hybrid Cloud	Hybrid Cloud	None	None	Runtime Benchmarking	Does not include orchestration or ML-based adaptation
[3] Henning & Hasselbring	Stream Frameworks as Cloud Microservices	Cloud	None	None	Scalability & Efficiency	Focuses on microservice-based deployment, not predictive routing
[12] Razzaq et al.	Hybrid Auto-Scaled Smart	Cloud	Rule-based Auto-scaling	Predictive Burst Model	Autoscaling Efficiency	Lacks multi-engine orchestration

	Campus System					and hybrid data routing
[13] Radhika & Sadasivam	Proactive-Reactive Autoscaling	Cloud	Dynamic Autoscaling	Statistical Prediction	Scaling Accuracy	Does not involve DAG-based orchestration or streaming pipelines
[14] Alsboui et al.	Distributed Intelligence in IoT	Edge-Cloud	Conceptual Routing	Theoretical AI Models	Architectural Taxonomy	Provides an IoT-oriented view, lacks implementation and orchestration validation
Proposed: HyScaleFlow	Spark + Flink with FlowGuard	Hybrid Edge-Cloud	Hybrid DAG (Airflow + Dagster)	XGBoost (Failure + Load Prediction)	Latency, Fault Tolerance, Cost, DAG Performance	Unified dataflow, real-time feedback, ML-driven preemptive orchestration

Table 12 provides a detailed comparative analysis between the proposed HyScaleFlow framework and five closely related works, selected from the reviewed literature. The comparison spans key dimensions including system architecture, orchestration strategy, machine learning integration, evaluation criteria, and distinctive contributions.

Ullah et al. [1] evaluated the performance of Spark, Flink, and Hadoop in hybrid cloud deployments. Their study is relevant in terms of benchmarking distributed engines, but it lacks orchestration logic and does not incorporate any adaptive or predictive mechanisms. HyScaleFlow builds on these foundational observations by integrating multi-engine orchestration with ML-guided decision-making.

Henning and Hasselbring [3] benchmarked stream processing frameworks deployed as microservices in cloud-only setups. While their work focuses on scalability and efficiency, it does not address hybrid cloud challenges or introduce any orchestration or ML components. In contrast, HyScaleFlow extends beyond pure benchmarking by actively managing real-time workloads across cloud and edge environments.

Razzaq et al. [12] introduced a hybrid auto-scaling approach using predictive models to anticipate burst workloads in a smart campus setting. Their use of ML for autoscaling aligns with the FlowGuard module in HyScaleFlow. However, their solution remains limited to cloud environments and lacks integration with distributed stream processing or DAG-based orchestration systems.

Radhika and Sadasivam [13] proposed proactive-reactive autoscaling using statistical forecasting. While this strategy shows promise for elasticity, it does not incorporate workflow-level orchestration or real-time feedback from system telemetry, both of which are central to HyScaleFlow's design. Moreover, their work does not involve task-level adaptation based on DAG semantics.

Alsboui et al. [14] explored distributed intelligence in IoT systems, proposing architectural concepts for edge-cloud integration and adaptive behavior. Although thematically similar to HyScaleFlow in terms of distributed architecture, their work is conceptual and lacks experimental validation, implementation details, and orchestration performance metrics.

In contrast to all these, HyScaleFlow distinguishes itself through its hybrid orchestration layer (Airflow + Dagster), real-time telemetry feedback via Prometheus, and ML-based orchestration via FlowGuard using XGBoost for failure prediction and load forecasting. It is the only framework among those compared that combines multi-engine stream processing, predictive adaptation, cost-aware resource efficiency, and complete DAG execution tracking in a hybrid edge-cloud environment.

5 Discussion

The rapid proliferation of real-time data-intensive applications across hybrid cloud and edge environments has led to the growing demand for scalable, responsive, and intelligent orchestration systems. Existing distributed stream processing frameworks, such as Apache Spark and Flink, offer strong processing capabilities but fall short in handling dynamic system behaviors, fault tolerance, and workload volatility without external orchestration layers. A review of the state-of-the-art reveals that while some

research has addressed performance benchmarking or autoscaling in isolation, there remains a clear gap in integrating predictive intelligence with real-time distributed data engineering across hybrid architectures. Most existing approaches either rely on static orchestration rules, lack fault anticipation, or fail to provide unified multi-engine coordination.

This gap necessitates the development of novel machine learning–driven orchestration strategies that can anticipate system bottlenecks, adapt DAG execution paths dynamically, and optimize resource usage without manual intervention. The proposed HyScaleFlow framework addresses this by introducing an intelligent orchestration module, FlowGuard, that leverages XGBoost models to predict both node-level failures and load surges. The architecture is uniquely designed to combine the strengths of Apache Airflow and Dagster, ensuring both scheduled and reactive orchestration, and enabling dynamic task routing between edge and cloud environments.

Experimental evaluations demonstrate significant improvements in system responsiveness, fault recovery, DAG completion rate, and cost efficiency. Results show that FlowGuard's predictive capabilities reduce task retry rates, improve uptime during failure scenarios, and enhance throughput per core, thereby overcoming key limitations of existing reactive and rule-based systems. The integration of ML within the orchestration pipeline proves critical in enabling scalable, fault-resilient, and resource-aware stream processing. The implications of this research are substantial for domains requiring continuous, intelligent dataflow management, including IoT, smart cities, and cyber-physical systems.

The existing approaches, e.g, Razzaq et al. [12] and Shahid et al. [33], which are more reactive in that they primarily provide fault- tolerance or burst- aware scaling, HyScaleFlow's FlowGuard enables predictive orchestration, where failures can be anticipated and the execution path of the DAG is dynamically adapted to reactively or proactively respond to the emergent failures. For instance, [12] uses a burst prediction model but it does not interoperate with a DAG-level orchestration over multi-engine sites. Similarly, Shahid et al. [33] also classify the fault-tolerance methods, but do not deploy a predictive recovery methods. HyScaleFlow on the other hand, reduces the recovery time up to 35% to 40% and task retry rate by 62.3% due to its strategy of employing two ML model. Moreover, Ullah et al. [1] Compare benchmark performance between Spark and Flink, but lack in orchestration and load prediction. HyScaleFlow extends this work by presenting its hybrid orchestration proposal and achieving +14.7% throughput with dynamic workloads. These comparisons also highlight the power of the PSOTA's ability to seamlessly integrate scalability, fault tolerance and preemptive orchestration beyond the state-of-the-art.

The current HyScaleFow implementation assumes that there is a trusted hybrid infrastructure where all communication between components (Kafka, Spark, Flink, FlowGuard) takes place on secure channels. But in real-world implementations, we have to deal with problems like exposed telemetry data, unauthorized access to orchestration APIs, and data leakage towards the edge-cloud boundary. Additional features in the future will include end-to-end encryption, role-based access control, and secure container orchestration to round out a holistic security architecture.

While the proposed framework addresses numerous limitations of prior art, Section 5.1 outlines the specific limitations of the present study.

5.1 Limitations of the study

While the proposed HyScaleFlow framework demonstrates significant improvements in orchestration intelligence and system efficiency, the current study has three notable limitations. First, FlowGuard's prediction models are trained offline and may require periodic retraining for evolving workloads. Second, the system was evaluated using a single dataset and fixed ingestion rates, limiting generalizability to diverse data sources. Third, while the framework supports hybrid orchestration, it does not yet include fine-grained cost-based task placement strategies across multiple cloud providers. Although both Spark and Flink were strategically chosen for edge and cloud tiers respectively according to processing patterns and latency/resource exchanges, an experimental investigation of contrasting role placement (e.g., Flink on edge, Spark in cloud) still represents a juicy subject for future research. The ablation analysis can also be used to better tune the task-to-resource mapping in hybrid deployment.

HyScaleFlow demonstrated competitive results up to 5,800 records/sec, and it is interesting to run further experiments (e.g., 10k or 50k records/sec) to determine its scaling limits and saturation point. We believe this is one of the key areas in need of future work in understanding how well-the programmability of the data plane translate into meaningful fault coverage at scale under various workloads.

Current pricing estimates were obtained using AWS on-demand pricing, to ensure consistent, reproducible benchmark conditions. In future work, a finer-grained cost sensitivity analysis with spot and reserved price-based costs will be considered to capture the operational variability in cloud economics and contribute to the deployability of deployments.

Future work can address these aspects to enhance adaptability, dataset diversity, and economic optimization in large-scale hybrid cloud deployments.

6 Conclusion and future work

This paper presented HyScaleFlow, a scalable and intelligent framework for real-time distributed data engineering in hybrid cloud environments. By integrating Apache Spark and Flink with a hybrid orchestration

strategy (Airflow and Dagster) and the FlowGuard ML module, the system effectively addresses critical challenges in fault tolerance, workload adaptation, and resource efficiency. Extensive experiments using the NYC Taxi Trip dataset demonstrated significant improvements in task completion rates, recovery time, throughput, and cost efficiency, validating the robustness and adaptability of the proposed methodology. The research fills existing gaps in the literature by introducing predictive, ML-driven orchestration into multi-engine streaming pipelines, offering a unified solution that extends beyond static rule-based models. It provides a modular, generalizable architecture suitable for real-time applications in smart cities, industrial IoT, and edge analytics. Future work will focus on enhancing FlowGuard's adaptability through online learning techniques and extending support for workload-aware, cost-optimized task placement across heterogeneous cloud providers. Additionally, evaluating the framework under diverse datasets and varying ingestion rates will further validate its generalizability. These advancements will position HyScaleFlow as a comprehensive orchestration solution for dynamic, large-scale, and cost-sensitive hybrid cloud ecosystems, building upon the strong foundation established in this study.

References

[1] Faheem Ullah, Shagun Dhingra, Xiaoyu Xia, and M. Ali Babar. (2024). Evaluation of distributed data processing frameworks in hybrid clouds. *Elsevier*. 224, pp.1-14. https://doi.org/10.1016/j.jnca.2024.103837

[2] Sivakumar Ponnusamy, and Pankaj Gupta. (2024). Scalable data partitioning techniques for distributed data processing in Cloud Environments: A Review. *IEEE*. 12, pp.26735 - 26746. DOI:10.1109/ACCESS.2024.3365810

[3] Sören Henning, and Wilhelm Hasselbring. (2024). Benchmarking scalability of stream processing frameworks deployed as microservices in the cloud. *Elsevier*. 208, pp.1-17. https://doi.org/10.1016/j.jss.2023.111879

[4] Reyazur Rashid Irshad, Shahid Hussain, Ihtisham Hussain, Jamal Abdul Nasir, Asim Zeb, Khaled M. Alalayah, Ahmed Abdu Alattab, Adil Yousif, and Ibrahim M. Alwayle. (2024). IoT-Enabled Secure and Scalable Cloud Architecture for Multi-User Systems: A Hybrid Post-Quantum Cryptographic and Blockchain-Based Approach Toward a Trustworthy Cloud Computing. *IEEE*. 11, pp.105479 - 105498. DOI:10.1109/ACCESS.2023.3318755

[5] Md. Motaharul Islam, and Zaheed Ahmed Bhuiyan. (2023). An integrated scalable framework for cloud and IoT based green healthcare system. *IEEE*. 11, pp.22266 - 22282. DOI:10.1109/ACCESS.2023.3250849

[6] Bayan H. Banimfreg. (2023). A comprehensive review and conceptual framework for cloud computing adoption in bioinformatics. *Elsevier*. 3, pp.1-13. https://doi.org/10.1016/j.health.2023.100190

[7] N. Sai Lohitha, and M. Pounambal. (2023). Integrated publish/subscribe and push-pull method for cloud based IoT framework for real time data processing. *Elsevier*. 27, pp.1-9. https://doi.org/10.1016/j.measen.2023.100699

[8] S¨ oren Henning, and WilhelmHasselbring. (2022). A configurable method for benchmarking scalability of cloud-native applications. *Springer*. 27(143), pp.1-42. https://doi.org/10.1007/s10664-022-10162-1

[9] Baldeep Singh, Randall Martyr, Thomas Medland, Jamie Astin, Gordon Hunter, and Jean-Christophe Nebel. (2022). Cloud based evaluation of databases for stock market data. *Springer*. 11(53), pp.1-17. https://doi.org/10.1186/s13677-022-00323-4

[10] Sabrine Khriji, Yahia Benbelgacem, Rym Chéour, Dhouha El Houssaini, and Olfa Kanoun. (1-28). Design and implementation of a cloud-based event-driven architecture for real-time data processing in wireless sensor networks. *Springer*. 78, p.3374–3401. https://doi.org/10.1007/s11227-021-03955-6

[11] Lei Chen, Jiacheng Zhao, Chenxi Wang, Ting Cao, Johnzigman, Haris Volos, Onurmutlu, Fang Lv, Xiaobing Feng, Guoqingharryxu, and Huimin Cui. (2022). Unified holistic memory management supporting multiple big data processing frameworks over hybrid memories. *ACM*. 39(1-4), pp.1-38. https://doi.org/10.1145/3511211

[12] Razzaq, M. A., Mahar, J. A., Ahmad, M., Saher, N., Mehmood, A., & Choi, G. S. (2021). Hybrid Auto-Scaled Service-Cloud-Based Predictive Workload Modeling and Analysis for Smart Campus System. IEEE Access, 9, 42081–42089. doi:10.1109/access.2021.3065597

[13] Radhika, E. G., & Sudha Sadasivam, G. (2021). A review on prediction based autoscaling techniques for heterogeneous applications in cloud environment. Materials Today: Proceedings, 45, 2793–2800. doi: 10.1016/j.matpr.2020.11.789

[14] Alsboui, T., Qin, Y., Hill, R., & Al-Aqrabi, H. (2021). Distributed Intelligence in the Internet of Things: Challenges and Opportunities. SN Computer Science, 2(4). doi:10.1007/s42979-021-00677-7

[15] Risco, S., Moltó, G., Naranjo, D. M., & Blanquer, I. (2021). Serverless Workflows for Containerised Applications in the Cloud Continuum. Journal of Grid Computing, 19(3). doi:10.1007/s10723-021-09570-2

[16] Hu, L., Zhang, F., Qin, M., Fu, Z., Chen, Z., Du, Z., & Liu, R. (2021). A Dynamic Pyramid Tilling

Method for Traffic Data Stream Based on Flink. IEEE Transactions on Intelligent Transportation Systems, 1–10. doi:10.1109/tits.2021.3060576

[17] Mohyuddin, S., & Prehofer, C. (2021). A Scalable Data Analytics Framework for Connected Vehicles Using Apache Spark. 2021 International Symposium on Electrical, Electronics and Information Engineering. doi:10.1145/3459104.3459156

[18] Ramalingeswara Rao, T., Ghosh, S. K., & Goswami, A. (2020). Mining user–user communities for a weighted bipartite network using spark GraphFrames and Flink Gelly. The Journal of Supercomputing. doi:10.1007/s11227-020-03488-4

[19] Van Dongen, G., & Poel, D. V. D. (2021). A Performance Analysis of Fault Recovery in Stream Processing Frameworks. IEEE Access, 9, 93745–93763. doi:10.1109/access.2021.3093208

[20] Ashiku, L., Al-Amin, M., Madria, S., & Dagli, C. (2021). Machine Learning Models and Big Data Tools for Evaluating Kidney Acceptance. Procedia Computer Science, 185, 177–184. doi: 10.1016/j.procs.2021.05.019

[21] Habib Mostafaei, Georgios Smaragdakis, Thomas Zinner, and Anja Feldmann. (2022). Delay-resistant geo-distributed analytics. *IEEE*. 19(4), pp.4734 - 4749. DOI:10.1109/TNSM.2022.3192710

[22] Salman Ahmed Shaikh, Hiroyuki Kitagawa, Akiyoshi Matono, Komal Mariam, and Kyoung-Sook Kim. (2022). GeoFlink: an efficient and scalable spatial data stream management system. *IEEE*. 10, pp.24909 - 24935. DOI:10.1109/ACCESS.2022.3154063

[23] Jianhao Chen, Zhuangzhuang Zhang, Xiyang Jiang, Jianpeng Huang, and Yifei Tong. (2022). Research on escalator data acquisition and transmission based on big data platform. *Elsevier*. 208, pp.532-538. https://doi.org/10.1016/j.procs.2022.10.073

[24] Habib Mostafaei, Shafi Afridi, and Jemal Abawajy. (2022). Network-aware worker placement for wide-area streaming analytics. *Elsevier*. 136, pp.270-281. https://doi.org/10.1016/j.future.2022.06.009 partitioning in Apache Flink and the cloud. *Springer*. 34(42), pp.1-15. https://doi.org/10.1007/s00138-023-01391-5

[25] Ana Almeida, Susana Brás, Susana Sargento, and Filipe Cabral Pinto. (2023). Time series big data: a survey on data stream frameworks, analysis and algorithms. Springer. 10(83), pp.1-32. https://doi.org/10.1186/s40537-023-00760-1

[26] Dimitrios Kastrinakis, and Euripides G.M. Petrakis. (2023). Video2Flink: real-time video partitioning in Apache Flink and the cloud. Springer. 34(42), pp.1-15. https://doi.org/10.1007/s00138-023-01391-5

[27] Weisi Chen, Zoran Milosevic, Fethi A. Rabhi, and Andrew Berry. (2023). Real-time analytics: Concepts, architectures, and ML/AI considerations. *IEEE*. 11, pp.71634 - 71657. DOI:10.1109/ACCESS.2023.3295694

[28] Guojian Xu, Mingyang Song, Zhenggang Leng, and Zhenhong Jia. (2023). Simulation Research on Fast Matching of Big Data Based on Spark. *IEEE*. 11, pp.32628 - 32635. DOI:10.1109/ACCESS.2023.3262989

[29] Moksud Alam Mallik, Nurul Fariza Zulkurnain, Sumrana Siddiqui, and Rashel Sarkar. (2024). The Parallel Fuzzy C-Median Clustering Algorithm Using Spark for the Big Data. *IEEE*. 12, pp.151785 - 151804. DOI:10.1109/ACCESS.2024.3463712

[30] Mohamed Yusuf Hassan. (2024). Applications of Bigdata Technologies in the Comparison of BMTD and ARIMA Models for the Prediction of Internet Congestion. *IEEE*. 12, pp.56642 - 56651. DOI:10.1109/ACCESS.2024.3389041

[31] Lisana Berberi, Valentin Kozlov, Giang Nguyen, Judith Sáinz-Pardo Díaz, Amanda Calatrava, Germán Moltó, Viet Tran, and Álvaro López García. (2025). Machine learning operations landscape: platforms and tools. *Springer*. 58(167), pp.1-37. https://doi.org/10.1007/s10462-025-11164-3

[32] Engin Zeydan, and Josep Mangues-Bafalluy. (2022). Recent advances in data engineering for networking. *IEEE*. 10, pp.34449 - 34496. DOI:10.1109/ACCESS.2022.3162863

[33] Shahid, M. A., Islam, N., Alam, M. M., Mazliham, M. S., & Musa, S. (2021). Towards Resilient Method: An exhaustive survey of fault tolerance methods in the cloud computing environment. Computer Science Review, 40, 100398. doi: 10.1016/j.cosrev.2021.100398

[34] Karthikeyan, L., Vijayakumaran, C., Chitra, S., & Arumugam, S. (2021). SALDEFT: Self-Adaptive Learning Differential Evolution Based Optimal Physical Machine Selection for Fault Tolerance Problem in Cloud. Wireless Personal Communications, 118(2), 1453–1480. https://doi.org/10.1007/s11277-021-08089-9

[35] Alaei, M., Khorsand, R., & Ramezanpour, M. (2020). An adaptive fault detector strategy for scientific workflow scheduling based on improved differential evolution algorithm in cloud. Applied Soft Computing, 106895. doi: 10.1016/j.asoc.2020.106895

[36] Nalini, J., & Khilar, P. M. (2021). Reinforced Ant Colony Optimization for Fault Tolerant Task Allocation in Cloud Environments. Wireless Personal Communications. doi:10.1007/s11277-021-08830-4

[37] A. U. Rehman, Rui L. Aguiar, and João Paulo Barraca. (2022). Fault-tolerance in the scope of cloud computing. *IEEE*. 10, pp.63422 - 63441. DOI:10.1109/ACCESS.2022.3182211

[38] Babak Taraghi, Hermann Hellwagner, and Christian Timmerer. (2023). LLL-CAdViSE: live low-latency cloud-based adaptive video streaming evaluation framework. *IEEE*. 11, pp.25723 - 25734. DOI:10.1109/ACCESS.2023.3257099

[39] Marios Fragkoulis, Paris Carbone, Vasiliki Kalavri, and Asterios Katsifodimos. (2024). A survey on the evolution of stream processing systems. *Springer*. 33, p.507–541. https://doi.org/10.1007/s00778-023-00819-8

[40] Cheng-Wei Ching, Xin Chen, Chaeeun Kim, Tongze Wang, Dong Chen, Dilma Da Silva, and Liting Hu. (2025). AgileDART: An Agile and Scalable Edge Stream Processing Engine. *IEEE*. 24(5), pp.4510 - 4528. DOI:10.1109/TMC.2025.3526143

[41] New York City Taxi and Limousine Commission (NYC TLC), 2024. *TLC Trip Record Data*. [online] Available at: https://www.nyc.gov/site/tlc/about/tlc-trip-record-data.page

[42] Guan, J. (2025). Enhanced Network Security Hybrid Cloud Workflow Scheduling Using Levy-Optimized Slime Mould Algorithm. Informatica, 49(18).

[43] Ilias, Shaik Mohammad, V. Ceronmani Sharmila, and V. Sathya Durga. "An Integrated Framework with Enhanced Primitives for Post-Quantum Cryptography: HEDT and ECSIDH for Cloud Data Security and Key Exchange." Informatica 49.11 (2025).

[44] Tang, Haili, and Zefeng Ding. "A Hybrid LSTM-Transformer Approach for State of Health and Charge Prediction in Industrial IoT-Based Battery Management Systems." Informatica 49, no. 22 (2025).