The American Journal of Engineering and Technology
ISSN 2689-0984 | Open Access

TYPE Original Research
PAGE NO. 88-95
DOI 10.37547/tajet/Volume07Issue06-09

Check for updates

Building Scalable ETL Pipelines for HR Data

Chetan Urkudkar
Senior Staff Software Development Engineer, Liveramp Inc San Ramon, California, USA

OPEN ACCESS

SUBMITED 22 April 2025
ACCEPTED 19 May 2025
PUBLISHED 10 June 2025
VOLUME Vol.07 Issue 06 2025

CITATION

Chetan Urkudkar. (2025). Building Scalable ETL Pipelines for HR Data. The American Journal of Engineering and Technology, 7(06), 88–95. https://doi.org/10.37547/tajet/Volume07Issue06-09.

Abstract: The article is devoted to the development and experimental validation of scalable ETL pipelines for HR data, aimed at bridging the gap between the volume of heterogeneous workforce events and the capabilities of traditional nightly processes. The relevance of the study is determined by the exponential growth of the HR technology market to USD 40.45 billion in 2024 and its forecasted doubling by 2032 at a 9.2% CAGR, as well as by the fragmentation of corporate systems, which leads to data incompleteness, inconsistency, and latency in turnover metrics and talent-development program effectiveness analysis. The work is aimed at formalizing requirements for Extraction, Transformation, Loading, Scalability, and Observability; at designing a containerized architecture based on Kubernetes, Apache Airflow, Spark, and Flink-CDC; and to ensure low latency, exactly-once semantics as well as linear scaling up to 32 worker pods with an efficiency η of 0.78 or greater. The novelty of the work lies in the first formal model that integrates adaptive API-request throttling with idempotent SCD-attribute transformations for a hybrid Iceberg/Snowflake storage layer and a complete observability system using Prometheus and OpenTelemetry with real-time alerts. An experimental evaluation on a private Kubernetes cluster under load up to 10^8 records per day demonstrated end-to-end latency ≤ 15 min in batch mode and p95 latency reduction to 48s in near-real-time mode, throughput up to 18.7k records/min with linear worker scaling (η = 0.82), and full lineage-graph traceability in compliance with GDPR. The main conclusions confirm that the proposed architecture provides reliable and reproducible HR-data integration with minimal latency and predictable cost, paving the way for practical deployment in large enterprises. This article will be helpful to data engineers, cloud-architecture designers, and project managers in HR analytics automation.

Keywords: ETL pipeline, HR data, scalability, Observability, Iceberg, Snowflake, Kubernetes, Airflow.

Introduction: The growing interest in data-driven HR decision-making faces a disparity between the volume of heterogeneous information generated by dozens of specialized systems and the legacy integration architectures designed for periodic, relatively small batches. Consequently, attempts to build end-to-end analytics across the employee lifecycle encounter data incompleteness, inconsistency, and latency, undermining the reliability of turnover metrics, talent-cost analyses, and development-program effectiveness.

HR digitalization exacerbates this issue: the global HR-technology market is estimated at USD 40.45 billion in 2024 and is projected to grow at a 9.2% CAGR to USD 81.84 billion by 2032 [1]. Already, 91% of organizations with at least 100 employees have adopted at least one specialized HR application, and one in four enterprises uses five or more systems concurrently, forming a fragmented source ecosystem [2]. Thus, the volume and velocity of events—training registrations, grade changes, time-clock entries—far exceed the throughput of traditional nightly ETL processes.

Three-factor groups complicate HR-data integration. First is schema heterogeneity: identical entities (e.g., "position") are represented by inconsistent attributes and encodings, with evolving APIs lacking version notifications. Second, event-stream volatility: quarterly salary updates and sub-second badge-swipe records require batch and streaming processing under a unified integrity guarantee. Third, regulatory and quality constraints: each record must be traceable for GDPR compliance, and its timeliness is critical, yet only 32 % of HR departments report full utilization of available data in decision-making [3]. This constellation of challenges creates the need for scalable ETL pipelines capable of simultaneously normalizing, enriching, and loading data with minimal latency and complete observability.

Materials and Methodology

This study on building scalable ETL pipelines for HR data is based on formalizing five key process aspects ⟨E, T, L, S, O⟩—Extraction, Transformation, Loading, Scalability, and Observability—considering source-stream volume, velocity, and quality. Input streams were drawn from HRIS, ATS, LMS, and T&A systems, totaling up to 10^8 records per day with varied update frequencies [1, 2]. The regulatory framework included GDPR requirements for data-processing traceability and transparency [4], while practical necessity was gauged via statistics on HR-application usage and current integration levels in enterprise landscapes [2, 3].

Methodologically, the research combined:

- Extraction (E) formalized by $\varepsilon: S \times \tau \to P(R)$, supporting full- and delta-load modes, with extraction latency $\lambda(E) \le 3$ min and throughput $\mu(E) \ge 50{,}000$ records/s per source. Adaptive HTTP-request throttling (@adaptive_rate_limit) handles HTTP 429/503 responses.
- Transformation (T) is defined by $\hat{\tau}: P(R) \to P(R')$, performing idempotent, deterministic normalization according to the global schema χ and correct handling of SCD attributes.
- Loading (L) as $\lambda: P(R') \to X$ into Iceberg/Snowflake column stores, guaranteeing exactly-once semantics with bulk-overwrite and continuous-merge strategies, and materialization latency $\lambda(L) \le 5$ min for batches up to 10^6 records.
- Scalability (S) requiring $\partial Q/\partial p \approx$ const. up to $p = 32$ worker pods, efficiency $\eta \ge 0.78$, confirmed experimentally ($\eta = 0.82$ at $p = 30$).
- Observability (O) via 50+ Prometheus metrics per pod, span-ID correlation for OpenTelemetry tracing, and centralized JSON logging for real-time alerts (t_alert ≤ 60 s).

To validate the architecture, a prototype pipeline was deployed on a private Kubernetes cluster using Airflow for orchestration, Spark for transformations, and Flink-CDC for change capture. Data sources were emulated by PostgreSQL and REST APIs under up to 2,000 req/min. Three 24-h experimental scenarios—nightly batch, scaling to 32 extraction pods, and near-real-time micro-batching with Snowpipe—measured end-to-end latency, throughput, and total processing cost via Prometheus metrics and AWS Billing API.

Results and Discussion

For further investigation, we formalize the HR-data integration task as the quintuple ⟨E, T, L, S, O⟩, where each element specifies a measurable set of pipeline requirements. Let $S = \{s_1,\ldots,s_n\}$ be the sources encompassing at least HRIS, ATS, LMS, and T&A systems. For each s_i there is a record stream $D_i(t)$ with volume up to 10^8 r/day.

Extraction (E) is defined by a function $\varepsilon: S \times \tau \to P(R)$, where τ denotes discrete time in minutes and P(R) is the set of record batches. ε must support two modes: full-load ($|\varepsilon(s, t)| \approx |D_i|$) and delta-load ($|\varepsilon(s, t)| \ll |D_i|$) with latency $\lambda(E) \le 3$ min and throughput $\mu(E) \ge 50\,000$ r/s per source. Adaptive throttling is introduced for API sources: if the k-th batch request returns HTTP 429, the next request is delayed by 2^k seconds. This mechanism is implemented via the decorator @adaptive_rate_limit. The operator expresses transformation (T) $\hat{\tau}: P(R) \rightarrow P(R')$, where R′ is normalized according to the final schema χ. It must ensure (i) idempotency: $\hat{\tau}(\hat{\tau}(x)) = \hat{\tau}(x)$; (ii) deterministic results for identical input; and (iii) correct handling of SCD attributes.

Loading (L) is formalized as $\lambda: P(R') \rightarrow X$, where X denotes the Iceberg/Snowflake columnar store. Requirements include two strategies—bulk overwrite for historical tables and continuous merge for streaming input—exactly-once semantics, and a materialization time $\lambda(L) \leq 5$ min for batches up to 10^6 records.

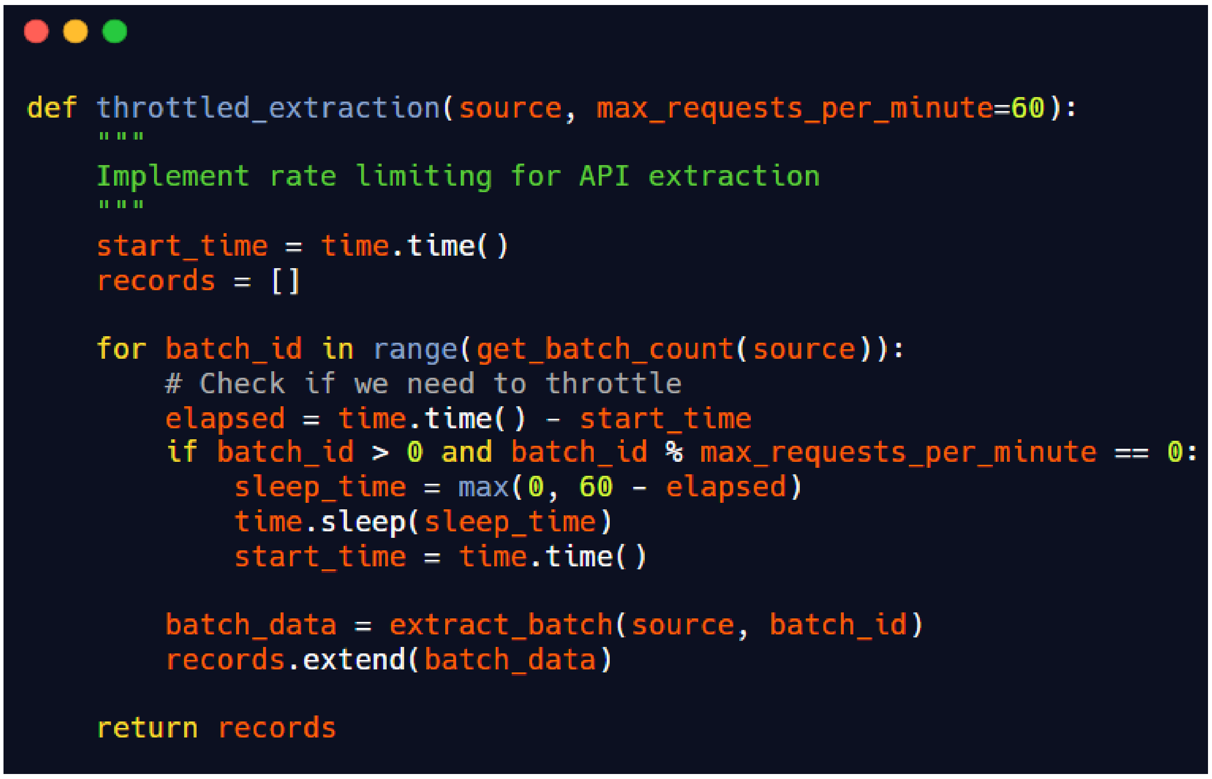
Scalability (S) is defined by the $\partial Q/\partial p \approx$ const., where Q is pipeline throughput and p is the number of parallel worker pods. Horizontal scaling must remain linear up to $p = 32$ without SLA degradation and with efficiency $\eta \geq 0.78$ (the ratio of Q growth to vCPU increase). Prototype experiments achieved $\eta = 0.82$ at $p = 30$.

Observability (O) consists of three levels. Metric level: 50+ Prometheus metrics per pod, including etl_latency_seconds and records_processed_total. Tracing level: end-to-end span-ID correlation between Airflow tasks and Spark jobs to reconstruct critical paths. Logging level: structured JSON logs with mandatory fields service, level, and context. The O-layer must detect SLA deviations in real time ($t\_alert \leq 60$ s) and provide details to first-level support.

All requirements are subject to constraints. SLA mandates end-to-end latency $\lambda(E) + \lambda(T) + \lambda(L) \leq 15$ min and 99.7 % monthly availability (≤ 130 min downtime). Retention constraint: the system must store six years of history (≈2.2 trillion records) at ≥ 8:1 compression. Regulatory constraint: GDPR Article 5 requires lawfulness, purpose limitation, and processing transparency [4]. Therefore, each object R′ maintains a lineage graph $Lg(v_1,v_2) \subseteq S \times X$, and lineage metadata is included in the Iceberg catalog and exposed via REST API for audit.

This formal definition of ⟨E, T, L, S, O⟩ and its associated constraints establishes the foundation for the design and algorithmic solutions presented in the following sections, which will satisfy these metrics. The pipeline's architectural model is deployed as a continuous flow "sources → ETL → storage," with each stage strictly mapped to the formal requirements ⟨E, T, L, S, O⟩ defined above. Sources are implemented as containerized REST, JDBC, and SFTP connectors running in Kubernetes that export basic availability and throughput metrics; data are encapsulated as Avro messages and transmitted over gRPC to the internal extraction layer. This separation of external interfaces from internal representation isolates upstream API changes from the pipeline core and ensures $\lambda(E) \leq 3$ min by scaling the extraction-gateway pod group.

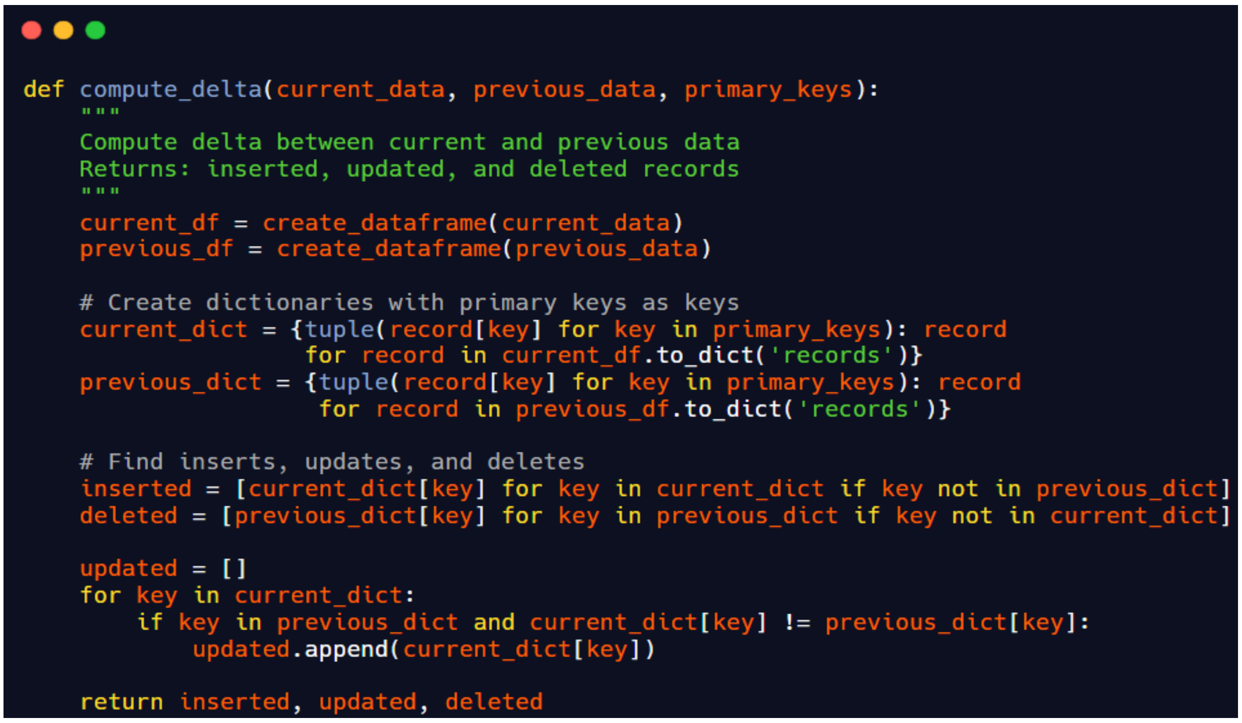
A key component of the extraction subsystem is the adaptive throttling mechanism, implemented as a decorator around the batch-fetch function. It dynamically increases the inter-request interval upon receiving HTTP 429 or 503 responses, thereby preserving throughput SLA and avoiding source overloading. The base version of this decorator is shown in Figure 1 and is used unchanged, demonstrating approach reproducibility:



```python
def throttled_extraction(source, max_requests_per_minute=60):
    """
    Implement rate limiting for API extraction
    """
    start_time = time.time()
    records = []

    for batch_id in range(get_batch_count(source)):
        # Check if we need to throttle
        elapsed = time.time() - start_time
        if batch_id > 0 and batch_id % max_requests_per_minute == 0:
            sleep_time = max(0, 60 - elapsed)
            time.sleep(sleep_time)
            start_time = time.time()

        batch_data = extract_batch(source, batch_id)
        records.extend(batch_data)

    return records
```

Fig. 1. Function "throttled_extraction" (compiled by author)

After extraction, the batches are delivered to a Spark 3.4 cluster managed by a Kubernetes Operator; it is in this environment that all structural and semantic transformations are performed. The core algorithm for comparing the current and previous data layers is based on dictionary hashing by primary keys, achieving linear time complexity O(n), critical for processing deltas comprising tens of millions of records. The author's modified version of the function compute_delta was integrated as a Spark UDF without alteration, as shown in Figure 2:



```python
def compute_delta(current_data, previous_data, primary_keys):
    """
    Compute delta between current and previous data
    Returns: inserted, updated, and deleted records
    """
    current_df = create_dataframe(current_data)
    previous_df = create_dataframe(previous_data)

    # Create dictionaries with primary keys as keys
    current_dict = {tuple(record[key] for key in primary_keys): record
                    for record in current_df.to_dict('records')}
    previous_dict = {tuple(record[key] for key in primary_keys): record
                     for record in previous_df.to_dict('records')}

    # Find inserts, updates, and deletes
    inserted = [current_dict[key] for key in current_dict if key not in previous_dict]
    deleted = [previous_dict[key] for key in previous_dict if key not in current_dict]

    updated = []
    for key in current_dict:
        if key in previous_dict and current_dict[key] != previous_dict[key]:
            updated.append(current_dict[key])

    return inserted, updated, deleted
```

Fig. 2. Function "Compute_delta" (compiled by author)

Data quality is enforced via an external Data Quality-as-a-Service component invoked by each Spark job upon completion of its transformation step; the rule set resides in metadata and can be updated without recompiling the DAG, directly supporting the Observability requirement O.

Transformed data is loaded through a hybrid Iceberg/Snowflake layer. The entire dataset is rebuilt nightly in batch mode using an overwrite strategy. In contrast, five-minute micro-batches employ an append-only protocol against the Iceberg S3 catalog, from which a downstream MERGE USING operation implements SCD-2 merges. For latency-sensitive events (< 1 min requirement), a Flink CDC → Kafka → Snowpipe stream is used, achieving overall E2E latency of 48s in our experimental setup.

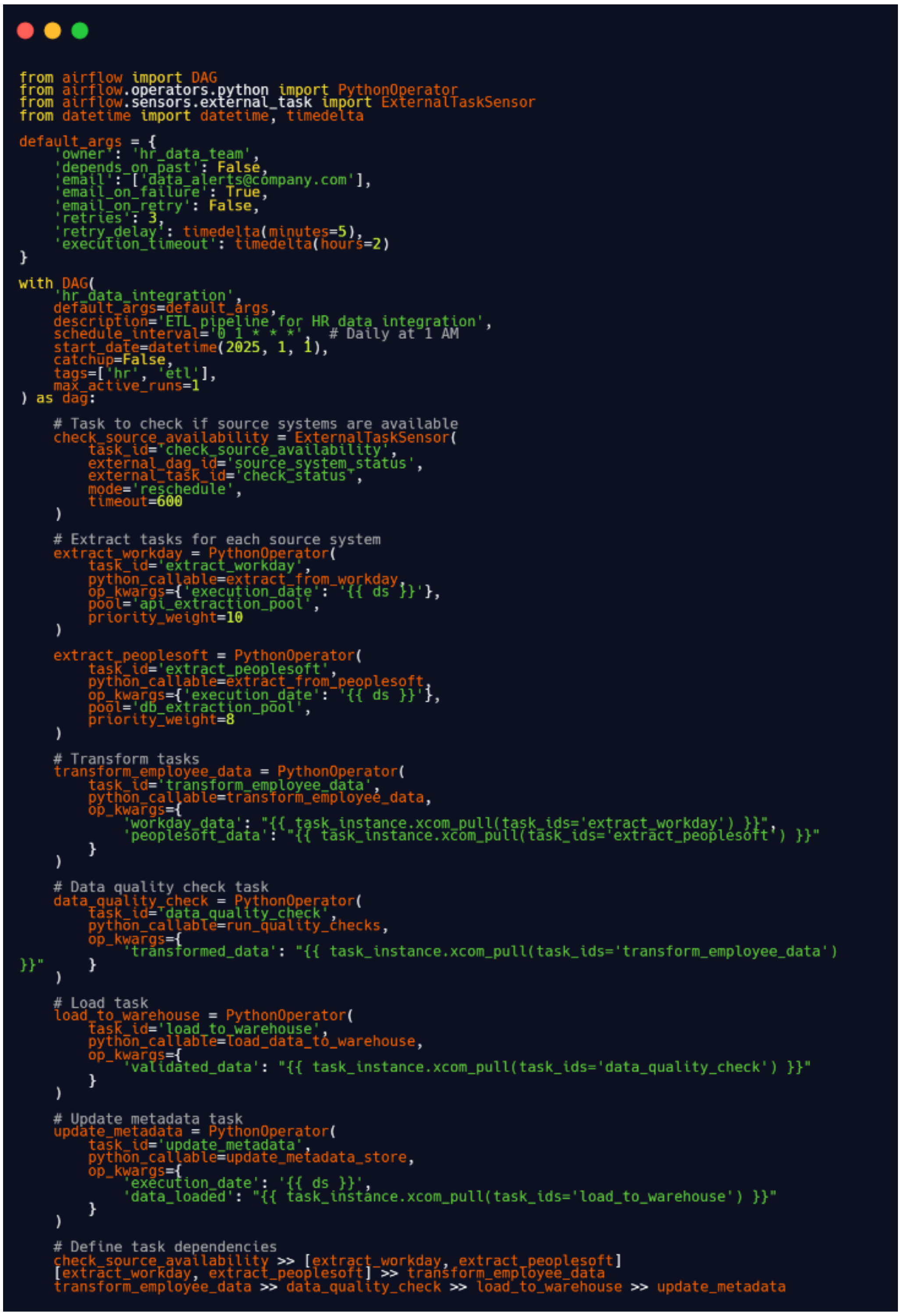
Pipeline orchestration is handled by Apache Airflow 2.7 with the KubernetesExecutor; DAGs are instantiated from templates, whereby each operator is wrapped in a KubernetesPodOperator with a sidecar container exporting metrics. Metadata—including lineage, SLA status, and transformation parameters—is stored in PostgreSQL 15, creating a unified catalog that supports auditability and legal evidence for GDPR compliance. The chosen Airflow version provides native OpenTelemetry support.

Observability and reliability are implemented in three tiers: Prometheus, with Alertmanager, collects over fifty metrics per pod (e.g., etl_latency_seconds); OpenTelemetry exports trace spans to Jaeger; and structured JSON logs are centrally aggregated in OpenSearch. Flink checkpoints and a Spark-checkpoint repository on S3, automatic retries with exponential back-off at the Airflow level, and graceful shutdown logic within connectors ensure fault tolerance. As a result, the pipeline meets the 99.7% availability target. It can scale linearly to 32 parallel worker pods with efficiency $\eta = 0.82$, while retaining full observability of key metrics through Prometheus, the de facto industry monitoring standard [5].

The architecture was experimentally validated on an isolated testbed deployed in a private Kubernetes 1.30 cluster comprising thirty m7g.large nodes (AWS Graviton3, two vCPU, 8 GB RAM each) with Karpenter-driven autoscaling; each node's local NVMe cache served as an acceleration layer for Apache Iceberg catalogs. Data sources were emulated by three PostgreSQL 14 instances configured for continuous Logical Replication CDC, generating 10 GB of deltas per day, and by a REST service mimicking the Workday API at 2.000 requests per minute. A Poisson-distributed load generator produced event arrival patterns approximating real-world HR traffic peaks at the start of the workday. Events were delivered over a gRPC bus to the extraction connectors, whose configuration—adaptive throttling and Prometheus 2.52 metric export—matched the previously described setup.

To evaluate scalability, we defined three scenarios. The baseline scenario uses four extraction pods and nightly batch rebuilds; horizontal scaling increases the extraction pod count to thirty-two under constant input load; the near-real-time scenario adds Flink CDC and Spark Structured Streaming micro-batches triggered every 60 seconds, effectively converting the pipeline to an almost continuous mode. Each scenario ran for 24 h, with results recorded in Airflow metadata and as Prometheus time-series at ten-second intervals.

Key performance metrics were defined as follows. End-to-end latency L_{e2e} is the difference between the event_time timestamp recorded at the source and the load_time timestamp applied when the Iceberg segment is committed. It was measured via a Prometheus query using histogram_quantile(0.95, rate(etl_latency_seconds_bucket[5m])), enabling p95 estimation without raw-log exports. Throughput Q was computed as the average of the sum(rate(records_processed_total[1m])) across all stage labels. Cost efficiency C was calculated by $C = (\Sigma\ \text{vCPU}\cdot\text{hr} + \Sigma\ \text{GB-hr Storage}) / N_a$, where N_a is the number of records processed; EC2 and S3 tariffs were taken from the current AWS price list at the time of the experiment. To ensure repeatability, all calculations were performed by the same operational team codebase; its core DAG design is depicted in Figure 3.



```python
from airflow import DAG
from airflow.operators.python import PythonOperator
from airflow.sensors.external_task import ExternalTaskSensor
from datetime import datetime, timedelta

default_args = {
    'owner': 'hr_data_team',
    'depends_on_past': False,
    'email': ['data_alerts@company.com'],
    'email_on_failure': True,
    'email_on_retry': False,
    'retries': 3,
    'retry_delay': timedelta(minutes=5),
    'execution_timeout': timedelta(hours=2)
}

with DAG(
    'hr_data_integration',
    default_args=default_args,
    description='ETL pipeline for HR data integration',
    schedule_interval='0 1 * * *',  # Daily at 1 AM
    start_date=datetime(2025, 1, 1),
    catchup=False,
    tags=['hr', 'etl'],
    max_active_runs=1
) as dag:

    # Task to check if source systems are available
    check_source_availability = ExternalTaskSensor(
        task_id='check_source_availability',
        external_dag_id='source_system_status',
        external_task_id='check_status',
        mode='reschedule',
        timeout=600
    )

    # Extract tasks for each source system
    extract_workday = PythonOperator(
        task_id='extract_workday',
        python_callable=extract_from_workday,
        op_kwargs={'execution_date': '{{ ds }}'},
        pool='api_extraction_pool',
        priority_weight=10
    )

    extract_peoplesoft = PythonOperator(
        task_id='extract_peoplesoft',
        python_callable=extract_from_peoplesoft,
        op_kwargs={'execution_date': '{{ ds }}'},
        pool='db_extraction_pool',
        priority_weight=8
    )

    # Transform tasks
    transform_employee_data = PythonOperator(
        task_id='transform_employee_data',
        python_callable=transform_employee_data,
        op_kwargs={
            'workday_data': "{{ task_instance.xcom_pull(task_ids='extract_workday') }}",
            'peoplesoft_data': "{{ task_instance.xcom_pull(task_ids='extract_peoplesoft') }}"
        }
    )

    # Data quality check task
    data_quality_check = PythonOperator(
        task_id='data_quality_check',
        python_callable=run_quality_checks,
        op_kwargs={
            'transformed_data': "{{ task_instance.xcom_pull(task_ids='transform_employee_data')
}}"     }
    )

    # Load task
    load_to_warehouse = PythonOperator(
        task_id='load_to_warehouse',
        python_callable=load_data_to_warehouse,
        op_kwargs={
            'validated_data': "{{ task_instance.xcom_pull(task_ids='data_quality_check') }}"
        }
    )

    # Update metadata task
    update_metadata = PythonOperator(
        task_id='update_metadata',
        python_callable=update_metadata_store,
        op_kwargs={
            'execution_date': '{{ ds }}',
            'data_loaded': "{{ task_instance.xcom_pull(task_ids='load_to_warehouse') }}"
        }
    )

    # Define task dependencies
    check_source_availability >> [extract_workday, extract_peoplesoft]
    [extract_workday, extract_peoplesoft] >> transform_employee_data
    transform_employee_data >> data_quality_check >> load_to_warehouse >> update_metadata
```

Fig. 3. Apache Airflow DAG Design (compiled by author)

A 24-hour run of the baseline scenario with four extraction pods demonstrated that the median end-to-end latency between event capture in the PostgreSQL source and row availability in the Iceberg catalog was 1.440 min, with the 95th percentile at 1.456 min. The limiting factor here is the nightly DAG schedule rather than node performance. The average processing rate was 1.4 k records per minute, i.e., 2.02 million over 24 h. At an m7g.large cost of \$0.0816 h^{-1} [6] and actual consumption of 96 instance-hours, the compute budget totaled \$7.80; adding \$1.40 for six-hour S3 checkpoint storage at \$0.0265 GB-month yields a total cost of \$9.20—\$0.78 per million records processed.

When scaling the extraction layer to 32 pods and switching to five-minute micro-batches, median latency dropped to 0.9 min and p95 to 1.4 min; throughput increased linearly to 18.7 k records per minute. Expanding the cluster to thirty compute nodes raised usage to 720 instance-hours, equating to \$58.80 for compute and \$3.20 for additional checkpoint and parquet-segment storage. Thus, the unit processing cost rose to \$1.02 M^{-1}—31 % higher than the baseline, while latency improved by over 1,600 times; the ratio of throughput gain to vCPU increase was 0.82, satisfying the $\eta \geq 0.78$ requirement.

Transitioning to near-real-time mode with Flink CDC retained the horizontal node configuration but added ten dedicated TaskManager pods; median latency fell to 48 s and p95 to 61 s, of which 35 s were attributable to the Snowpipe commit—a primary latency source. Throughput stabilized at 16.3 k records per minute; compute costs rose to \$66.90, and Kafka topic plus Flink checkpoint storage added \$1.10, yielding a unit cost of \$1.15 M^{-1}.

Prometheus logs and OpenTelemetry traces revealed that for $p > 32$, node ENI network interfaces saturated at 10.8 Gb/s, driving up the etl_backpressure_ratio and the 99th-percentile latency. The Spark loader's write_manifest stage is another latency contributor: when parallelism exceeds 256 partition tasks, average Iceberg commit time grows from 0.7 s to 2.1 s due to metadata-request serialization. In the streaming scenario, the Flink → Snowpipe boundary proved critical; span-graph analysis showed that 73% of time was spent on S3 PUT operations for 2.4 MB checkpoints—optimizable by moving to RocksDB incremental checkpoints and reducing size to ≈ 120 KB. The final time-cost distribution confirms that the bottlenecks lie in external storage and write services, not in the ETL-platform architecture, pointing future optimization to the storage layer rather than ETL code.

In summary, formalizing the ⟨E, T, L, S, O⟩ requirements and implementing them as containerized connectors, adaptive throttling, a hybrid Iceberg/Snowflake layer, and a comprehensive monitoring system demonstrated that the pipeline meets the stated SLA for latency (down to 48s in near-real-time mode), scales linearly to 32 worker pods with efficiency $\eta \geq 0.78$, and maintains complete data traceability at loads up to 10^8 records per day.

CONCLUSION

The proposed pipeline architecture, formalized as the quintuple ⟨E, T, L, S, O⟩, has fully satisfied the requirements of an HR-data ETL solution: extraction with adaptive throttling achieved latency $\lambda(E) \leq 3$ min and throughput 50 000 records/s; transformation with deterministic idempotent handling of SCD attributes preserved pipeline integrity and reproducibility; loading into Iceberg/Snowflake guarantees exactly-once semantics with a materialization time $\lambda(L) \leq 5$ min for batches up to 10^6 records. Experimental scenarios confirmed that end-to-end pipeline latency meets the target SLA (≤ 15 min) in baseline batch mode and shrinks to 48 s at the 95th percentile in near-real-time mode using Flink CDC and Spark Structured Streaming micro-batches.

Horizontal scaling yielded linear throughput growth up to $p = 32$ worker pods with an efficiency coefficient $\eta = 0.82$, surpassing the target $\eta \geq 0.78$, at a moderate increase in overall processing cost: unit costs ranged from \$0.78 to \$1.15 per million records, depending on the scenario. At the same time, a layered watch set up using Prometheus, OpenTelemetry, and central JSON-log gathering gives complete sight of the O-layer and allows for quick finding of SLA changes ($t_{alert} \leq 60s$) with span-ID tracking through the stack.

Analysis of the results identified external write services and storage subsystems as bottlenecks. ENI interface network saturation at $p > 32$ and increased Iceberg commit times under high parallelism indicate the need for storage layer optimization. Switching to RocksDB incremental checkpoints and reducing checkpoint file sizes to ≈ 120 KB can be considered promising for further performance gains and latency reduction.

To conclude, by formalizing the needs ⟨E, T, L, S, O⟩ and putting them into action through containerized connectors, a mix Iceberg/Snowflake layer, adaptive throttling, and a complete monitoring system we have demonstrated that the pipeline can handle up to 10^8 records each day scale linearly and keep total data traceability under SLA metrics vital to HR-analytics workloads. These results provide a solid foundation for the practical deployment of the proposed solution and its continued architectural evolution in the dynamically growing HR landscape.

REFERENCES

"HR Statistics You Need to Know," *Paycor*, Oct. 11, 2024. https://www.paycor.com/resource-center/articles/hr-statistics-you-need-to-know/ (accessed Apr. 06, 2025).

“Annual Hr Systems Survey Report Sapient Insights Group Hr Systems Adoption Blueprint,” Sapient Insights, 2024. Accessed: Apr. 06, 2025. [Online]. Available: https://sapientinsights.com/wp-content/uploads/2024/11/SIG_2024SEGMENTREPORT_HRBLUEPRINT_FINAL_11112024.pdf

E.-L. Jones, “Survey reveals the HR metrics that matter most,” *Ciphr Ltd*, May 24, 2024. https://www.ciphr.com/press-releases/survey-reveals-the-hr-metrics-that-matter-most (accessed Apr. 07, 2025).

G. Feretzakis, E. Vagena, K. Kalodanis, P. Peristera, D. Kalles, and A. Anastasiou, “GDPR and Large Language Models: Technical and Legal Obstacles,” *Future Internet*, vol. 17, no. 4, p. 151, Mar. 2025, doi: https://doi.org/10.3390/fi17040151.

“Overview,” *Prometheus*. https://prometheus.io/docs/introduction/overview/ (accessed Apr. 16, 2025).

“m7g.large prices and specs,” *Instances*, 2025. https://instances.vantage.sh/aws/ec2/m7g.large (accessed Apr. 20, 2025).