

Article

A Metadata-Driven Execution Model for Unified Integration and Management of Heterogeneous IoT Data Sources

Marios Koniaris ¹, Danae Spentzou ¹, Max Friedemann ², Helmut Mischo ², John Soldatos ³
and Georgios Kouzas ^{3,*}

¹ Division of Computer Science, School of Electrical and Computer Engineering, National Technical University of Athens, Iroon Polytechniou 9, Zographou Campus, 15780 Athens, Greece; mkoniari@central.ntua.gr (M.K.); el20237@mail.ntua.gr (D.S.)

² Institute of Mining and Special Civil Engineering, Chair Underground Mining Methods, TU Bergakademie Freiberg, Akademiestraße 6, 09599 Freiberg, Germany; max.friedemann@mabb.tu-freiberg.de (M.F.); helmut.mischo@mabb.tu-freiberg.de (H.M.)

³ Innov-Acts Limited, Kolokotroni 6, Nicosia 1101, Cyprus; jsoldat@innov-acts.com

* Correspondence: gkouzas@innov-acts.com

Abstract

Mining operations generate continuous sensor data across heterogeneous repositories with no unified access layer. Existing integration platforms either require centralizing data into new infrastructure or demand extensive pipeline reconfiguration when sources change. We present a metadata-driven execution model in which integration behavior is resolved at runtime from executable metadata rather than encoded in static workflows, preserving existing infrastructure while enabling unified access across heterogeneous repositories. An Asset Cataloging registry stores executable specifications, including connector identifiers, connection parameters, and routing rules, which select and invoke the appropriate connector at runtime without workflow coding or redeployment. Evaluation on large-scale real mining sensor datasets spanning heterogeneous formats (JSON, CSV, Parquet) and repositories (Kafka, MongoDB, external REST APIs) confirmed zero message loss and bit-exact binary reconstruction across all scenarios under at-least-once delivery with idempotent writes. Connector dispatch overhead fell below the 1 ms measurement resolution, confirming that integration latency is dominated by storage I/O rather than orchestration cost. Following evaluation, four pilot sites deployed the platform in production, spanning from active underground operations to post-mining waste management, under the EU Horizon Europe MINE.IO project, demonstrating viability at industrial scale.

Keywords: data integration; metadata-driven; heterogeneous data sources; industrial IoT; mining; infrastructure preservation



Academic Editor: Abdun Mahmood

Received: 20 May 2026

Revised: 3 July 2026

Accepted: 13 July 2026

Published: 17 July 2026

Copyright: © 2026 by the authors.

Licensee MDPI, Basel, Switzerland.

This article is an open access article distributed under the terms and conditions of the [Creative Commons Attribution \(CC BY\)](https://creativecommons.org/licenses/by/4.0/) license.

1. Introduction

Industrial Internet of Things (IoT) applications generate continuous data streams from sensors and cameras across multiple sectors from healthcare to manufacturing [1,2]. Mining faces distinct challenges. Sensor networks at mining sites capture temperature, pressure, vibration, and environmental parameters at high frequencies, supplemented by surveillance cameras and geospatial data from drones. This continuous monitoring directly impacts operational safety, efficiency, and decision-making [3]. Equipment failures carry severe consequences. A single mine hoist failure lasting four days resulted in documented economic losses of 22.794 million RMB [4].

Yet this data volume creates integration difficulties. Mining data exhibit substantial heterogeneity, originating from multiple vendors and stored across disparate repositories with different data models, access protocols, and storage formats [3]. Each site may employ different technologies and separate IT infrastructures. The absence of unified standards complicates this further, making information utilization difficult [5].

Fragmentation also affects operations directly as real-time decisions become harder. Organizations struggle to detect deviations promptly, assess productivity accurately, or comply with regulations. Clean, well-structured data remain scarce, inhibiting AI and machine learning applications that could improve productivity and safety [6,7]. Digital transformation in mining faces a steep learning curve, requiring new skills in data management and analytics [8]. Valuable data sit unused in isolated silos, preventing integrated view of all production aspects.

These integration challenges extend beyond mining to smart cities, healthcare, and manufacturing, wherever heterogeneous IoT data span multiple repositories [9,10]. Prior work covers parts of this challenge but leaves a critical gap. Big data platforms like Hadoop or Spark [11,12] require consolidating data into unified storage, displacing existing repositories. Integration tools like Apache NiFi [13] and Kafka Connect [14] preserve existing systems but require custom workflow authoring for each integration scenario, encoding routing logic in static configurations; adding a new data source requires modifying and redeploying pipeline definitions. IoT interoperability standards such as W3C WoT [15] and FIWARE NGSI-LD [16] address semantic interoperability but not data movement across heterogeneous repositories. Metadata catalogs like Apache Atlas and AWS Glue excel at data discovery and governance but stop at documentation; they tell you *what data exist* and *where they live*, not *how to access them* or *how to move them* across systems.

Existing approaches address these capabilities individually but not in combination within a single, validated platform for the industrial IoT: runtime connector selection from a queryable metadata registry (no pipeline redeploy required when sources change), infrastructure preservation, and unified support for both real-time streaming and batch operations. This gap motivates the following research question: *Can executable metadata drive heterogeneous IoT integration without workflow reconfiguration or infrastructure replacement?*

Where these systems encode routing logic inside workflow graphs, connector configurations, or pipeline definitions, we externalize orchestration into *executable metadata*: integration behavior is resolved at runtime by querying a registry, rather than embedded in application workflows. We realize this execution model as a metadata-driven integration platform, in which metadata become the orchestration layer rather than replacing data repositories or requiring complex workflows. An Asset Cataloging registry captures each source's location, format, access parameters, and data characteristics, while a connector pattern provides standardized integration adapters for different storage technologies. Separating data source descriptions (what, where) from integration mechanisms (how) enables runtime connector selection and extensible integration (Figure 1).

Our solution emerged from the MINE.IO project, funded under EU Horizon Europe. MINE.IO is developing a Digital Mine 4.0 ecosystem; our platform provides the data integration infrastructure. We designed the platform for unified data access despite diverse storage technologies, across six operational pilot sites in Germany, Poland, Greece, Finland, and Portugal.

This work builds on European IoT integration research. An IIoT platform for Industry 4.0 manufacturing was developed by the FASTEN project (Horizon 2020) [17], addressing vertical, horizontal, and end-to-end integration through dual-lane architecture (FIWARE and Apache). Federated architecture reference models with metadata-driven federation layers were established by the IoT-A project (FP7) [18]. Our platform builds on these

concepts, extending metadata from discovery only to runtime executable specifications driving connector selection and data movement, while preserving existing infrastructure as validated in earlier approaches.

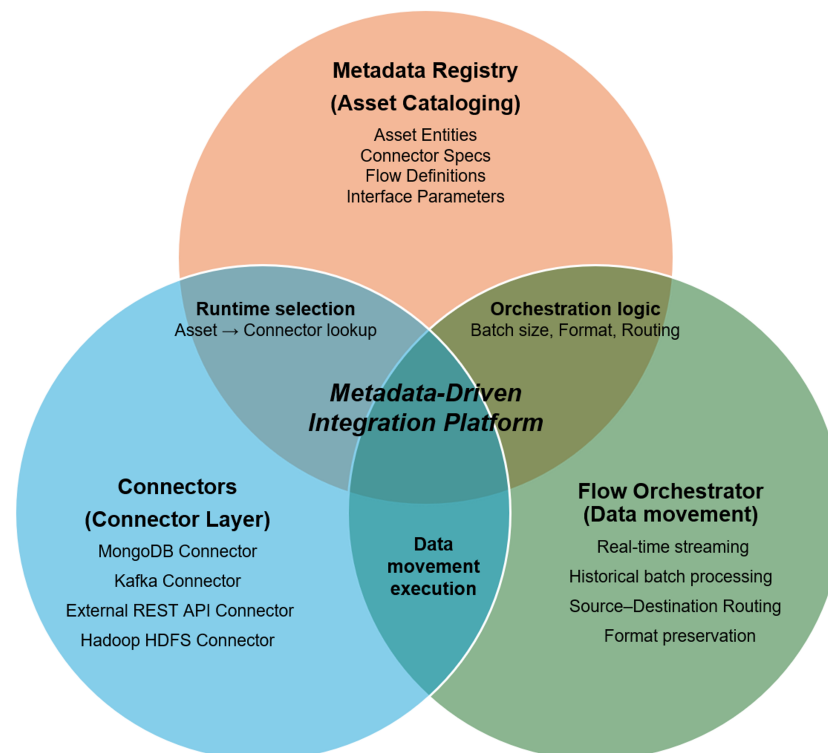


Figure 1. Conceptual framework of the platform. Three components interact, the metadata registry (Asset Cataloging), the connector layer, and flow orchestration, with connectors selected at runtime by querying the registry for connection parameters.

We evaluated our platform using real mining sensor data and simulated streaming scenarios across heterogeneous formats (JSON, CSV, Parquet, and binary archives) and multiple data sources including Apache Kafka, MongoDB, and external REST APIs. Evaluation validated complete data accuracy and format preservation. Following successful evaluation, four pilot sites deployed the platform in production, providing ongoing evidence of viability. This multi-site validation, spanning from active underground operations to post-mining waste management, strengthens findings beyond single-organization deployments.

Our platform is distinguished by combining three properties that existing systems provide only in isolation: (i) metadata that are executable rather than merely descriptive—stored specifications drive connector invocation at runtime, not just documentation; (ii) infrastructure preservation—no consolidation or replacement of existing repositories is required; and (iii) unified support for both real-time streaming and batch operations across heterogeneous storage backends. The following contributions operationalize this combination:

- (i) We introduce a metadata-driven integration platform, a concrete and deployed implementation in which an Asset Cataloging registry stores metadata as executable specifications (connector identifiers, connection parameters, and routing rules) that drive integration operations at runtime without workflow coding or infrastructure replacement. Unlike static pipeline configurations such as NiFi flows or Kafka Connect instances, these specifications are queried per operation at runtime; registering a new data source in Asset Cataloging makes it immediately accessible without redeployment or pipeline modification.

- (ii) We propose a connector pattern with runtime selection, where metadata drive dynamic connector invocation, eliminating hardcoded integration logic. The standardized four-operation interface (retrieve, file_retrieve, store, file_upload) maintains flexibility across heterogeneous repositories while supporting extensibility through metadata-only registration.
- (iii) We provide industrial scale validation, evaluating the approach on 137.2 million real mining sensor records (13.98 GB) across heterogeneous formats and repositories, demonstrating data accuracy, format preservation, and successful production deployment at four pilot sites.

The rest of this paper is organized as follows. Section 2 reviews related work, and Section 3 defines system requirements derived from the MINE.IO pilot context. Section 4 details the platform architecture and core components. Section 5 describes implementation specifics, and Section 6 presents evaluation methodology and results, while Section 7 discusses findings, generalizability, and design implications. Finally, Section 8 concludes with a summary and future research directions.

2. Related Work

We review existing approaches to data integration, metadata management, big data processing and IoT integration, highlighting gaps our platform addresses.

2.1. Big Data Processing Platforms

Big data platforms have evolved to handle volume, velocity, and variety [19]. Apache Hadoop enables distributed processing through HDFS and MapReduce [11], but batch orientation can limit real-time use. Apache Spark improves performance via in-memory processing and micro-batching [12,20], while Flink provides true event-at-a-time streaming [21,22]. For data movement, Sqoop handles database-to-Hadoop bulk transfers, and NiFi offers flexible dataflow automation.

Despite their processing power, all require custom integration code for each source. Few if any employ metadata orchestration; integration logic lives in code or workflows rather than queryable specifications. Infrastructure replacement or complex configuration remains unavoidable. Our platform, by contrast, integrates with Hadoop HDFS as an existing storage backend via WebHDFS, so pilot sites already operating Hadoop clusters connect without migrating other systems; HDFS serves as one storage endpoint among several rather than a consolidation target.

2.2. Data Integration Approaches

Traditional ETL/ELT patterns extract, transform, and load data into centralized warehouses [23]. Data virtualization creates logical views over distributed sources, while federated systems allow unified querying across heterogeneous databases [18].

These approaches struggle in IoT contexts. ETL/ELT assumes batch processing and centralized storage, conflicting with real-time requirements. Virtualization adds query latency and poorly supports streaming. Federated systems handle structured databases but struggle with diverse IoT formats and protocols. Timing alignment across asynchronous heterogeneous sensors remains particularly challenging [9].

Multi-repository systems demonstrate *polyglot persistence*: multiple specialized storage technologies optimized for different data characteristics. MusQ [24] and BigDAWG [25] unify queries across RDBMS, NoSQL, and HDFS but focus on reading rather than data movement. Manufacturing integration architectures [26] use message queuing to preserve legacy systems, sharing our infrastructure preservation principle and implementing Enterprise Integration Patterns for modular composition, concepts we adopt through the

connector pattern. SOA-based middleware solutions [27] demonstrate heterogeneous data fusion for IoT environments, while real-time stream processing systems [28] address semantic annotation challenges for continuous sensor data. Data lake architectures [29] employ metadata models for IoT dataset discovery across multi-source repositories.

2.3. Metadata Management and Data Catalogs

Commercial and open-source metadata catalogs (Alation, Collibra, Apache Atlas, AWS Glue, Google Dataplex) provide centralized repositories for data discovery and governance [30]. Schema registries manage data structure evolution in streaming contexts. Both address data discovery and governance but do not drive integration operations; metadata remain descriptive rather than executable.

Metadata-driven integration manifests through complementary approaches: classification methods generate metadata for unlabeled sources [31], ontology-based frameworks harmonize heterogeneous vocabularies using standards like OWL and RDF, and centralized metadata catalogs orchestrate runtime integration. Our platform focuses on the latter, providing executable metadata specifications for dynamic connector selection.

Domain-specific systems demonstrate metadata orchestration. Healthcare sensor platforms use metadata to drive sensor discovery and data retrieval [32], metadata-centric frameworks allow unified access across heterogeneous information sources [33], and disaster response systems employ metadata as operational specifications [34]. Scientific data platforms [35] employ centralized metadata-based integration with RESTful APIs for multidisciplinary data access. Linked stream middleware [36] enriches sensor streams with semantic descriptions for SPARQL-based querying. Recent work on metadata-driven ingestion patterns [37] demonstrates dynamic pipeline orchestration through metadata specifications, while real-time semantic annotation systems [38] manage metadata freshness for IoT sensor streams. Across these cases, metadata shift from descriptive to prescriptive, paralleling our Asset Cataloging's dual role.

Recent data governance approaches emphasize provenance and data contracts. Reproducibility-oriented systems (e.g., TableVault) capture execution parameters and metadata, while governance-enforcing patterns (e.g., ETLT/ELTL++) apply obligations during ingestion. Dataspace connectors (e.g., Eclipse Dataspace Connector, International Data Spaces) provide cataloging with usage control and identity management for cross-organizational data sharing, and streaming integration frameworks (e.g., Kafka Connect with Schema Registry) offer plugin-based integration with schema validation. Their concerns are complementary: provenance, governance enforcement, sovereignty, and schema management. Our platform focuses on technical integration and format preservation, positioning metadata as orchestration rather than governance. Future work could integrate these governance capabilities within our metadata-driven framework.

2.4. IoT Platform Standards and Interoperability

IoT interoperability initiatives address standardized data models and protocols. FIWARE [16,39], Arrowhead [40], and IIRA [41,42] define architectural patterns emphasizing semantic interoperability and modular composition. The W3C Web of Things (WoT) [15] provides Thing Descriptions as machine-readable metadata for device interaction models. Ontology-based approaches [43] support cross-domain IoT platforms with consistent sensor data acquisition specifications. Semantic architectures for pervasive computing [44], and Web-of-Things-based approaches [45] demonstrate SPARQL-based mechanisms for transparent device discovery and access across heterogeneous ecosystems. REST API-based approaches [46] use interworking proxies for protocol translation conceptually similar to our connector pattern but targeting service composition rather than data integration.

These frameworks establish semantic interoperability foundations. They define what data should look like, not how to integrate existing diverse systems with established data models and storage technologies.

The reviewed work reveals complementary strengths but leaves a critical gap for mining digital transformation [5,8]. Existing platforms either require infrastructure replacement (Hadoop MapReduce consolidation, Spark), demand complex configuration (NiFi), focus on governance over execution (metadata catalogs), or address semantic rather than technical integration (IoT standards). Unlike NiFi's flow-graph reconfiguration, Kafka Connect's connector-centric runtime management, or Camel's DSL-based routing, none provide per-asset executable metadata specifications for runtime connector selection without workflow changes. To our knowledge, no platform for mining IIoT combines infrastructure preservation with metadata-driven orchestration of data movement across heterogeneous repositories.

Table 1 makes this gap concrete: across the dimensions central to IIoT data integration, each existing approach satisfies only a subset, and none combines them all.

Table 1. Comparison of integration and data-lake approaches against the proposed platform across dimensions central to IIoT data integration.

Solution	Infrastructure Preservation	No Workflow Coding	Executable Metadata	Heterogeneous Backends	New-Backend Extensibility	Stream + Batch	Data Transformation
Apache NiFi	Yes	No	No	Partial	Custom code	Yes	Yes
Kafka Connect	Yes	No	No	Kafka-centric	Custom code	Yes	Partial
Apache Flink	No	No	No	Partial	Custom code	Yes	Yes
Traditional ETL/ELT	No	No	No	No	Custom code	No	Yes
Data lake + catalog (AWS Glue/Atlas)	No	No	Descriptive	Partial	Custom code	Partial	Partial
IoT standards (FIWARE/W3C WoT)	Yes	No	No	No	Custom code	No	No
Proposed platform	Yes	Yes	Yes	Yes	Metadata	Yes	No

Partial = supported with constraints or additional configuration. New-backend extensibility: "Custom code" = adding a storage backend requires writing new integration code; "Metadata" = it is added by registering connector metadata, with no platform changes.

Our platform addresses this gap by treating Asset Cataloging as an execution layer rather than a documentation layer. Stored metadata drive connector selection and invocation at runtime, eliminating workflow coding. The connector pattern standardizes access across heterogeneous repositories without requiring replacement, and a unified architecture supports both streaming and batch operations across existing systems.

3. Requirements Analysis

The platform design emerged from direct collaboration with mining operation pilot sites within the project. In the following we present the functional and non-functional requirements of our work.

3.1. Requirements Gathering

Requirements emerged from a systematic analysis across six operational pilot sites in Germany, Poland, Greece, Finland, and Portugal, where we conducted collaborative workshops with 27 consortium partners. Cross-site analysis revealed common integration challenges: each site had invested in specialized storage technologies, yet data exhibited substantial heterogeneity (time series readings, images, binary files, structured documents). The platform needed to support both real-time monitoring and historical analysis. Infrastructure replacement proved economically and operationally infeasible. These constraints drove platform design toward metadata-driven integration that preserved existing investments while providing unified access.

3.2. Data-Related Requirements

Data-Type Versatility. Mining operations generate diverse data types. Sensor networks produce numerical time series data at varying frequencies (0.1 ms to 100 ms intervals, i.e., up to 10,000 Hz). Visual monitoring systems capture images and video from fixed cameras and drones. Geospatial systems track location and spatial mapping. Technical documentation, production logs, and maintenance records exist in both structured (CSV, JSON) and unstructured (PDF, CAD files) formats. The platform must accommodate this variety without imposing format standardization. Data transformation before storage must not be required, as different data types serve distinct purposes and some need preservation in original formats for compliance or technical reasons.

Heterogeneous Storage Infrastructure. Each pilot site invested in storage technologies suited to operational needs, including MongoDB for document storage, time series databases for sensor data, Apache Kafka for event streaming, and Hadoop HDFS for distributed large-scale file storage. Sites also maintain external enterprise systems accessible via REST APIs for technical documentation and specifications. Integration with existing repositories and external systems must occur without mandating consolidation or replacement. Support for multiple, heterogeneous data sources simultaneously is required, while still providing unified data access. In practice, pilot sites fell into three categories: sites with existing storage infrastructure (PLM systems accessible via external REST APIs, existing MongoDB deployments) where preservation was mandatory; sites with no existing database, where MongoDB was standardized as the default deployment; and sites with large-scale file storage requirements, where Hadoop HDFS was selected. MongoDB was the most widely available storage backend across project partners, which also motivated its selection as the reference connector implementation for evaluation in Section 6.

Source-Type Routing and Format Preservation. Data from different sources must be routed to appropriate repositories based on source characteristics and site-specific preferences. Routing decisions must account for data type and format requirements. Data ingested in a specific format must remain reconstructible in that format; downstream applications must receive data in expected formats without requiring manual conversion.

Mining Safety and Diagnostics. Mining operations impose safety-critical requirements beyond generic IoT monitoring. Mine hoist systems require real-time monitoring of mechanical components, braking systems, and operational parameters [4]. Fault diagnosis must integrate real-time sensor streams with historical maintenance data and regulatory specifications. This integration spans structured databases, document repositories, and real-time streams. Supporting such safety-critical scenarios is a core platform requirement, as equipment failures in mining carry severe economic and human consequences.

3.3. Platform Functional Requirements

Discoverability. Users and external systems must have access to available data assets through search and browsing capabilities. Each data source must provide metadata descriptions including data type, format, update frequency, and ownership. The system must support queries such as “find all temperature sensors in mine section A” or “list all point cloud datasets from Pilot 2” through search and API queries.

Data Operations. The platform must support three core operations across all data sources: (i) *Retrieve*: Extract data from repositories with filtering capabilities by time range or specific attributes. (ii) *Upload*: Insert new data, supporting both individual records and bulk batch imports. (iii) *Flow*: Automate continuous data movement with configurable parameters for batch size, frequency, and format. These operations must function uniformly regardless of underlying storage technology (e.g., relational or NoSQL database, Kafka, or external APIs).

3.4. Platform Non-Functional Requirements

Reliability and Fault Tolerance. Industrial monitoring requires zero data loss. The system must handle network interruptions without data loss. When connections fail during data streaming or upload operations, automatic retry mechanisms must recover without manual intervention. Detailed error logging is required to support rapid diagnosis. Operators must have visibility into failure conditions, including which connector failed, what operation was attempted, and what error occurred.

High Performance and Real-Time Support. Mining safety systems may require sub-second response times. When sensors detect anomalous readings, data must flow immediately to alert systems. The system must support high-frequency data streams without blocking or data loss; multiple sensors can produce millions of records per minute, and the platform must handle these volumes efficiently. Historical analysis requires batch operations on large datasets. Analysts must be able to retrieve months of sensor data for trend analysis or model training without impacting real-time operations.

Maintainability and Modularity. The platform must provide a modular architecture with defined REST APIs for each component [47]. Modularity must support independent development, testing, and deployment. Connectors must not affect the core platform. Adding support for new data sources must be possible through standard connector interfaces without modifying existing code. The system must provide monitoring capabilities offering visibility into operations, data-flow statistics, and connector health to support maintenance and performance tuning.

3.5. Security and Governance Requirements

Authentication and Authorization. The platform must support role-based access control (RBAC) at multiple levels: asset access, connector invocation, and flow management. Authentication mechanisms must integrate with existing enterprise identity providers through standard protocols (OAuth 2.0, LDAP, SAML). User credentials must not be stored directly within the platform; instead, integration with centralized identity management systems guarantees consistent authentication across organizational boundaries.

Data Protection. All data transmission between components must support encryption in transit (TLS 1.3). Data at rest encryption remains the responsibility of underlying storage repositories (MongoDB, Kafka), following the principle of separation of concerns. The platform metadata layer should minimize direct storage of sensitive data; where deployment policy requires, connection credentials can be externalized through environment variables or secret management systems rather than embedded in the catalog. This design limits credential exposure in metadata catalogs without restricting operational flexibility.

Metadata-Based Access Policies. Access control decisions use metadata attributes: pilot site assignments, data classifications, and connector permissions. For example, a user from Pilot A can only create flows involving assets assigned to Pilot A. This metadata-driven approach to authorization aligns with the platform's core design philosophy, where metadata serve not only as documentation but as executable policy specifications. Asset metadata include ownership, access restrictions, and data sensitivity classifications, supporting fine-grained access control without hardcoded permission lists.

Audit and Compliance. All integration operations (data retrieval, upload, flow creation) must generate audit logs capturing timestamp, user identity, operation type, assets involved, and outcome. These logs support regulatory compliance (GDPR Article 30 for processing activities) and security forensics. Immutable audit trails support retrospective analysis of data access patterns, both for compliance verification and incident investigation.

Threat Model Considerations. The platform assumes a trusted network environment (typical for the industrial IoT within enterprise boundaries). External threats (DDoS,

injection attacks) are addressed at the network perimeter through firewalls and intrusion detection systems. Internal threats (unauthorized data access, privilege escalation) are mitigated through RBAC and audit logging. The threat model acknowledges that metadata catalogs themselves become high-value targets, as they provide a complete map of data assets; therefore, catalog access requires elevated privileges and enhanced monitoring. Future work will address fine-grained data-level access control, attribute-based policies, and integration with Security Information and Event Management (SIEM) systems for advanced threat detection.

3.6. Requirements to Platform Mapping

The above requirements collectively motivate the platform's three core architectural pillars, described in detail in Section 4. Table 2 summarizes how each requirement group is addressed by a specific platform mechanism.

Table 2. Mapping of system requirements to the platform mechanisms that address them.

Requirement	Platform Mechanism	Section
Data-type versatility and format preservation	Connector operation interface (API) handles both structured data and binary files; format is preserved end-to-end without transformation	Section 4.4
Heterogeneous storage infrastructure (preserve existing/standardize new)	Connector pattern: existing repositories (PLM/API, MongoDB) are preserved via dedicated connectors; MongoDB deployed as default where no database exists; Hadoop HDFS for big data. All cases handled through the same connector abstraction without platform core changes	Section 4.4
Source-type routing	Asset Cataloging registry: connector selected at runtime from asset metadata; routing requires no hardcoded logic	Section 4.2
Discoverability	Asset Cataloging CRUD API: assets queryable by type, pilot, format, and ownership	Section 4.2.3
Data operations (Retrieve/Upload/Flow)	Standardized four-operation interface (retrieve, file_retrieve, store, file_upload) plus flow management for automated pipelines	Sections 4.4 and 4.2.2
Reliability and fault tolerance	At-least-once delivery with automatic retry and exponential backoff; Kafka offset management prevents data loss on failure	Section 4.4
High performance and real-time support	Kafka streaming for real-time ingestion; configurable batch size for latency/throughput tradeoff; concurrent flow support	Section 6
Maintainability and modularity	Microservice architecture; new connectors registered without modifying platform core	Sections 4.4.2 and 5
Security and governance	RBAC, TLS 1.3, audit logs; metadata-based access policies enforced at asset, connector, and flow levels	Sections 3.5 and 5.5

4. Metadata-Driven Architecture

We present an architectural overview of the platform design, the Asset Cataloging component, and the entity-relationship metadata model.

4.1. Architectural Overview

The platform organizes functionality into five layers (Figure 2), each with distinct responsibilities. This follows established architectural decomposition principles [48] to allow independent evolution of each tier.

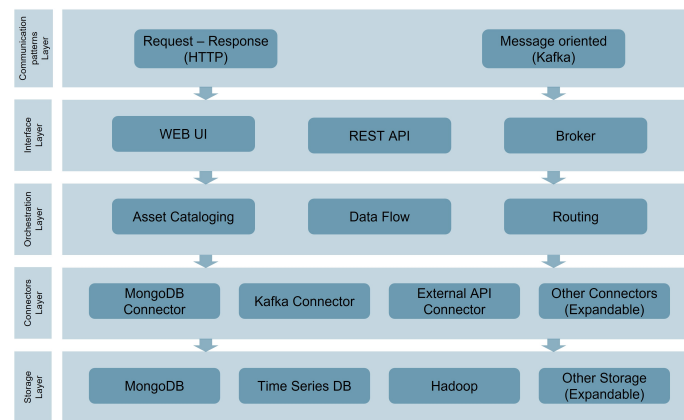


Figure 2. Five-layer platform architecture showing logical separation of concerns. Each layer has defined responsibilities: Communication (protocols), Interface (API/UI), Asset Cataloging (metadata orchestration), Connector (integration adapters), and Storage (heterogeneous repositories).

Figure 3 positions the platform within the broader IIoT ecosystem, illustrating how the five-layer architecture supports integration across Edge, Information, and Analytics domains while addressing crosscutting concerns of connectivity, interoperability, and distributed data management.

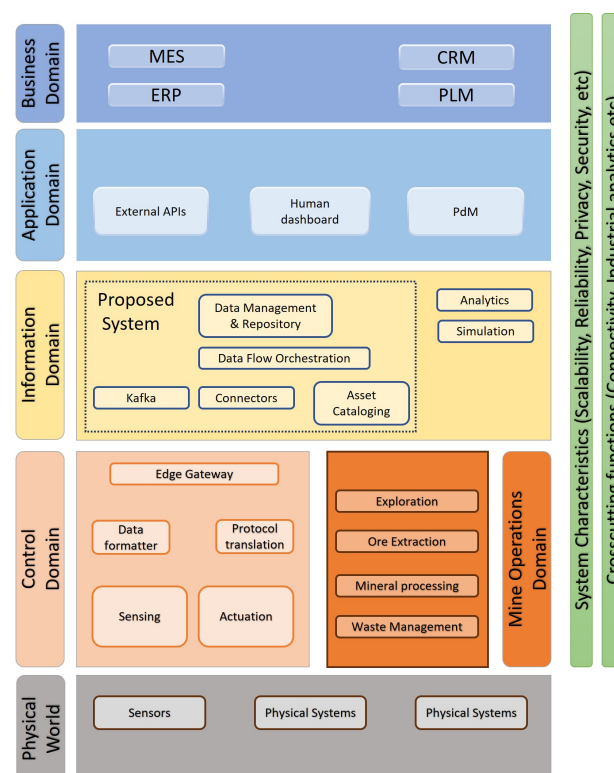


Figure 3. Platform positioning within the IIoT reference architecture. The proposed system occupies the Information domain (Asset Cataloging, Connectors, Flow Orchestration, Data Management), between edge control below and the Application and Business layers above. The Mine Operations Domain spans all layers, and the system characteristics and crosscutting functions shown apply across the full stack.

4.1.1. Communication Layer

The topmost layer establishes two communication paradigms matching different data interaction patterns. HTTP-based request response communication supports synchronous operations like data retrieval, metadata queries, and configuration management. Message-oriented communication via Apache Kafka handles asynchronous, high-volume data streams where producers and consumers operate independently. This dual approach accommodates both interactive user operations and continuous IoT data flows.

4.1.2. Interface Layer

The interface layer exposes platform capabilities through two mechanisms. A RESTful API provides programmatic access for external systems, allowing automated integration with analytics tools, visualization platforms, or other services. The endpoints follow REST principles [49], using standard HTTP methods (GET, POST, DELETE) and returning data in JSON format. A web-based user interface built on this API lets operators manage assets, configure connectors, create data flows, and monitor operations without programming.

4.1.3. Asset Cataloging Layer

This layer implements the metadata registry that drives runtime integration: metadata stored here are an executable specification, not merely descriptive documentation. Asset Cataloging maintains a MongoDB database storing metadata for all data sources, connectors, and data flows. When a user requests data from an asset, the registry provides the connector identifier, connection parameters, and data format rules needed to construct the integration call dynamically.

4.1.4. Connector Layer

The connector layer contains integration adapters that translate between the platform's unified interface and specific storage technologies. Each connector implements a standard interface with four operations (retrieve, file_retrieve, store, file_upload) but encapsulates technology specific connection logic, authentication mechanisms, and data format handling. The platform invokes connectors dynamically based on metadata specifications rather than hardcoded logic.

4.1.5. Storage Layer

The bottom layer comprises heterogeneous data repositories employed by pilot sites: MongoDB databases for flexible document storage, Kafka topics for event streaming, Hadoop HDFS for distributed large-scale file storage, and time-series databases for sensor data. The platform also integrates with external enterprise systems via REST API connectors, reaching distributed data sources without modifying or replacing existing infrastructure.

Figure 4 shows how these layers interact during a data operation, with five numbered steps tracing a request through the architecture. A user action in the Web UI issues a call to the REST API (*step 1*). The REST API forwards the request to the Data Flow Orchestrator within the Platform Backend (*step 2*). The Orchestrator queries Asset Cataloging to resolve the connector identifier and connection parameters for the target asset (*step 3*). Armed with these metadata-derived specifications, the Orchestrator invokes the appropriate connector (*step 4*). The connector translates the platform's unified operation into a technology-specific call and accesses the target storage repository (*step 5*). The architecture follows Service-Oriented Architecture (SOA) principles [47,50], with services communicating through well-defined interfaces: REST APIs for synchronous requests, message queues (Kafka) for asynchronous streaming, and direct database connections for data access. Each component operates independently, so development, testing, and deployment can proceed in parallel.

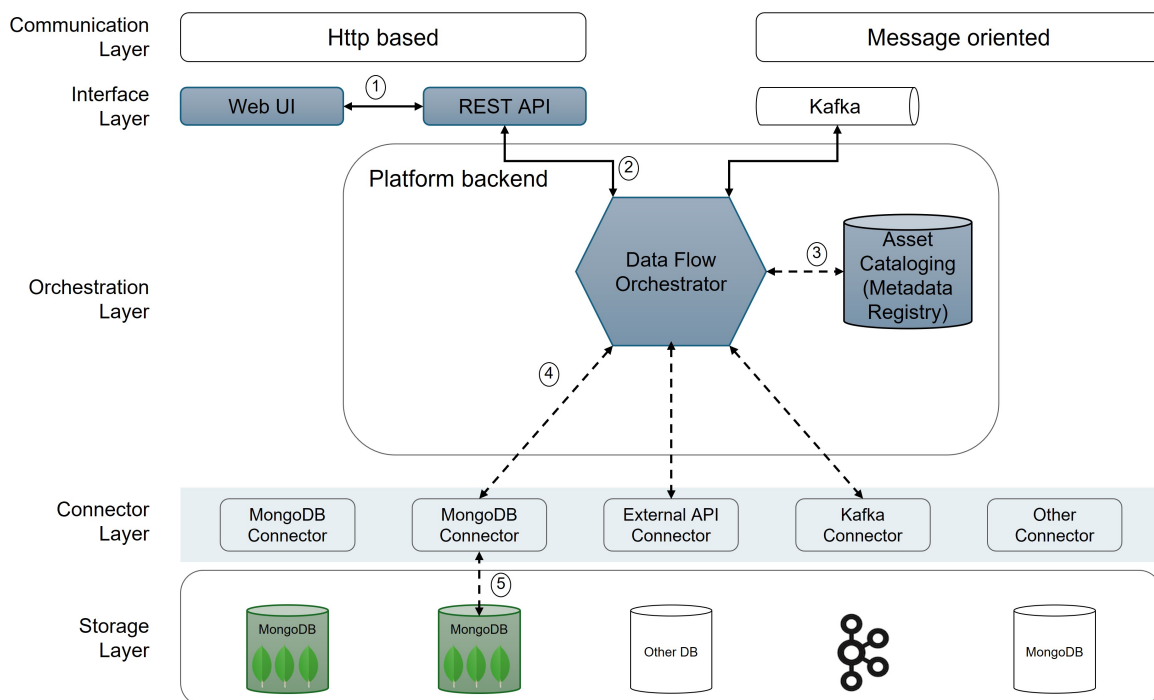


Figure 4. System architecture and the runtime path of a single operation across the five layers: Web UI to REST API to Data Flow Orchestrator, which queries Asset Cataloging for connector metadata, invokes the selected connector, and accesses the target repository.

4.2. Asset Cataloging: The Metadata Registry

Asset Cataloging stores and serves platform metadata through a custom implementation built for the MINE.IO context. This design choice reflects the requirement for executable metadata (connector identifiers, connection parameters, routing rules) beyond what discovery-oriented standards provide. The model is conceptually aligned with established interoperability standards without adopting their schemas, namespaces, or API bindings. The asset corresponds to `dcatalog:Dataset` (capturing name, description, type, format, and organizational assignment), and the connector corresponds to `dcatalog:DataService` (specifying endpoint URLs and operation capabilities); full DCAT adoption would nonetheless require an extension, since Flow, which defines automated data movement between assets, has no direct DCAT equivalent. W3C WoT Thing Descriptions describe device affordances and protocol bindings that align with the asset's `interface_*` connection parameters, but WoT operates at the device description level and does not cover inter-asset flow orchestration. Semantic frameworks such as FIWARE NGSI-LD are treated as upstream data producers: a Kafka topic carrying NGSI-LD entities is registered as a standard asset, and the platform ingests it without requiring any knowledge of the edge deployment that produced it. Where these standards target discovery and governance, our custom REST API drives execution: stored metadata specify the connector, connection parameters, and format rules, resolved when an operation runs.

4.2.1. Metadata Entity Schemas

Three core entity schemas structure the metadata model (Table 3). Assets describe data sources with model metadata (description, type, format), pilot assignment, and interface metadata (connector reference and connection parameters). Connectors specify integration capabilities through endpoint arrays. Flows define automated data movement between assets with transfer parameters.

Table 3. Core metadata entity schemas. Each entity serves distinct purposes: assets describe data sources, connectors specify integration mechanisms, and flows configure automated data movement.

Entity	Key Fields	Purpose
Asset	model_* (name, description, type, format) pilot interface_* (connector_id, parameters)	Data source description Organizational assignment Integration specifications
Connector	connector, description endpoints_* (name, method, url)	Connector identification Operation specifications
Flow	source_id, dest_id batch_size, format, status	Asset relationships Transfer configuration

4.2.2. Operations Flow

Data retrieval illustrates how metadata drive the complete integration sequence. When a request arrives, the platform queries Asset Cataloging for the target asset's metadata, extracts the connector identifier and connection parameters, retrieves the corresponding connector's endpoint specifications, and constructs the integration call dynamically. No hardcoded routing logic or configuration files are involved; the platform derives all integration details at runtime through metadata traversal. The connector handles technology-specific connection management and returns data in their original format. This traversal pattern, from asset metadata through connector specification to dynamic invocation, applies uniformly to all four operations.

4.2.3. Asset Discovery and Management

The Asset Cataloging exposes CRUD (Create, Read, Update, Delete) operations via REST API (Table 4), supporting automated asset registration: new sensors can self-register upon deployment by posting their metadata, making them immediately discoverable and accessible through the platform.

Both Asset and Connector entities expose identical CRUD API patterns (Table 4), with paths using either /cataloguing/asset or /cataloguing/connector. The Connector API additionally provides a type query endpoint (GET /cataloguing/connector/Type) for discovering available connector types. Adding new storage technologies requires only registering connector metadata (no platform core changes needed). Schema evolution follows the same additive principle: new optional metadata fields (e.g., data owner, retention policy, domain-specific attributes) can be introduced without migrating existing asset registrations, since the document-oriented backing store tolerates partial documents. The connector interface contract (retrieve, file_retrieve, store, file_upload) represents the primary stability boundary; changes to these signatures would require re-implementing existing connectors. In practice, this interface remained stable across all six pilot requirements and four production deployments.

Table 4. Cataloging API pattern for asset and connector management. Both entities follow this structure, with paths using either /cataloguing/asset or /cataloguing/connector.

Operation	Method	Description
List all	GET/cataloguing/{entity}	Retrieve all registered entities
Create	POST/cataloguing/{entity}	Register new entity with metadata
Get by ID	GET/cataloguing/{entity}/{id}	Retrieve specific entity metadata
Delete	DELETE/cataloguing/{entity}/{id}	Remove entity from registry

When a flow activates, the platform traverses metadata relationships: it resolves the source and destination connectors by following each asset reference in the flow record. It extracts connection parameters from both assets, invokes the source connector's retrieve endpoint to fetch data, and invokes the destination connector's store endpoint to write data. Configurable batch size and format parameters control transfer behavior; data transfers in batches without content modification, with the format parameter specifying the connector handler for each payload type. The flow activation mechanism follows a stateless orchestration pattern: integration logic is externalized to the Asset Cataloging layer, so high availability and fault tolerance hold independent of individual flow state. The Flow Management API covers the flow lifecycle—creation, monitoring, stop, and deletion—exposed to end users through the Platform API.

4.3. Entity-Relationship Metadata Model

The metadata model comprises three core entities, Asset, Connector, and Flow, connected by two relationships (Figure 5). Multiple assets share a single connector type, and one asset can participate in multiple flows; for example, a single Kafka topic can feed several MongoDB collections simultaneously for raw storage, aggregated summaries, and long-term archival.

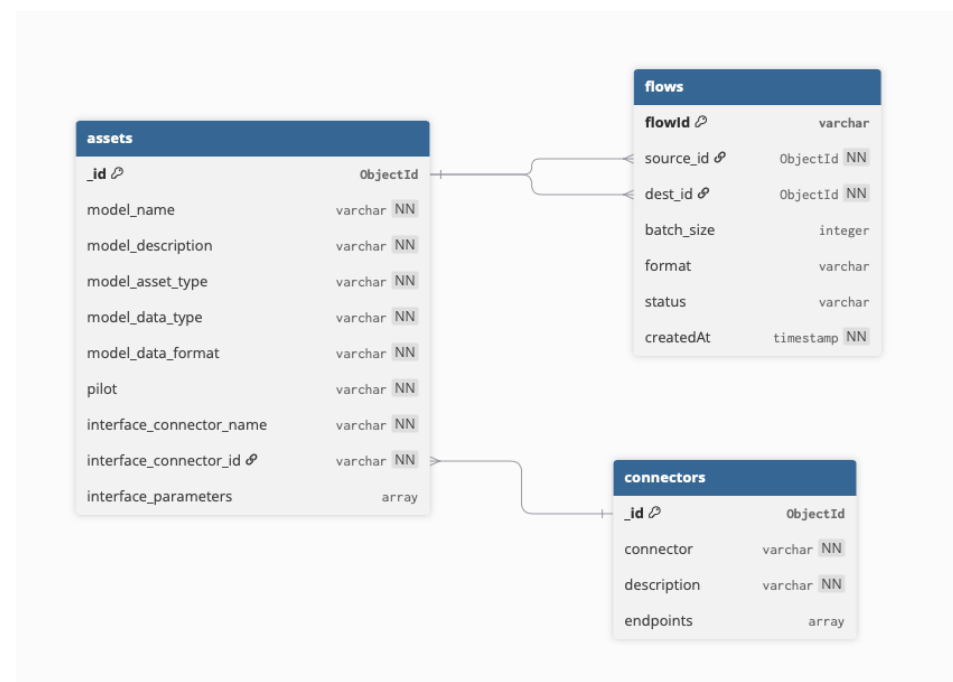


Figure 5. Entity-Relationship diagram of the metadata model: three entities (Asset, Connector, Flow) with Asset → Connector (N:1) and Flow → Asset (N:1) relationships. Gray boxes show key attributes.

Relationships exist through metadata references rather than hardcoded dependencies. To add a new asset, simply register its metadata. To add a connector type, assets reference it by updating the connector reference field. To create a flow, specify source and destination IDs; the platform resolves integration details through metadata traversal.

Asset Cataloging uses MongoDB as its metadata store; this instance is part of the platform infrastructure and is not registered as a managed asset. MongoDB was selected for three reasons: (1) its document-oriented model accommodates heterogeneous metadata schemas across diverse asset types without enforcing a rigid relational schema and supports additive schema evolution, where new optional fields such as data owner, retention policy, or domain-specific attributes can be introduced without migrating existing asset

registrations; (2) its rich JSON querying enables the metadata traversal operations that drive runtime connector selection, such as resolving all assets of a given connector type assigned to a specific pilot; and (3) it was the most widely available storage backend across MINE.IO project partners, with several pilot sites already operating MongoDB instances, making it the natural choice for both platform infrastructure and the reference connector implementation evaluated in Section 6. High-availability deployment of this MongoDB instance is an operations concern; active flows continue if the registry becomes temporarily unavailable, but new flow creation requires recovery.

The design embodies declarative integration: metadata specify what to do rather than code implementing how to do it. This lets non-programmers configure integrations through metadata management.

4.4. Connector Pattern for Extensible Integration

While metadata orchestrate integration decisions, connectors implement the actual data access. The connector pattern provides uniform access to diverse storage technologies while preserving technology-specific capabilities.

Unlike classical Adapter or Mediator patterns, which bind adapters at compile or configuration time, this connector pattern selects integration logic at runtime. The platform queries the registry for the asset's connector reference, retrieves connector endpoint specifications, and constructs the call dynamically, eliminating hardcoded assignments so connectors can be swapped by updating metadata alone without redeployment. The standardized four-operation interface preserves a consistent abstraction while metadata parameters accommodate technology-specific extensions; evaluation confirms negligible overhead at industrial scale (Section 6). The connector layer implements at-least-once delivery through automatic retry with exponential backoff, and idempotent writes prevent duplicate records on retry.

Each connector exposes four standardized operations (Table 5), accepting `asset_id` for metadata-driven parameterization.

Table 5. Standard connector interface specification enabling uniform platform integration.

Operation	Description
retrieve	Retrieve structured data with filtering
file_retrieve	Retrieve files or ZIP archives
store	Store structured data with batch insertion
file_upload	Upload files (single or multiple)

We implemented four connectors:

- (i) **MongoDB** supports structured data via batch insertion and binary files via Base64 encoding with format preservation.
- (ii) **Kafka** handles streaming via consumer groups, backpressure control, and batch aggregation.
- (iii) **Hadoop HDFS** supports large-scale distributed file storage via WebHDFS, integrating with Hadoop-based data-lake deployments.
- (iv) **External REST API** provides dynamic authentication and protocol translation via parameterized configuration for integrating third-party enterprise systems.

MongoDB is the reference connector implementation because it covers both operation types: structured data operations (retrieve, upload) and binary file operations (file_retrieve, file_upload). It implements the standard four-operation interface through the endpoints in Table 6.

Table 6. MongoDB connector endpoint implementations.

Endpoint	Method	Description
/api/data/retrieve	GET	Retrieve JSON/CSV with pagination
/api/file/retrieve	GET	Retrieve binary files as ZIP
/api/data/upload	POST	Batch insert structured data
/api/file/upload	POST	Upload with Base64 encoding

4.4.1. Runtime Invocation Flows

We present each runtime flow at two levels of abstraction, so that the same metadata-driven traversal can be read first in principle and then in practice. At the abstract level, Figures 6 and 7 trace the upload and retrieval paths as generic UML sequence diagrams: every request traverses the Backend, Asset Cataloging, and Connector before reaching the storage repository, independent of any particular transport. Figures 8 and 9 then instantiate exactly these two paths with Kafka as the concrete streaming transport, showing that the abstract skeleton holds once a real technology is plugged in. The two levels share the same actors, named here by role: the *Flow Management* microservice exposes the flow lifecycle API and implements the *Flow Orchestration* logic, while its runtime component (the *Flow Engine*, also called the Data Flow Orchestrator) resolves the source and destination connectors from Asset Cataloging and executes active data transfer jobs.

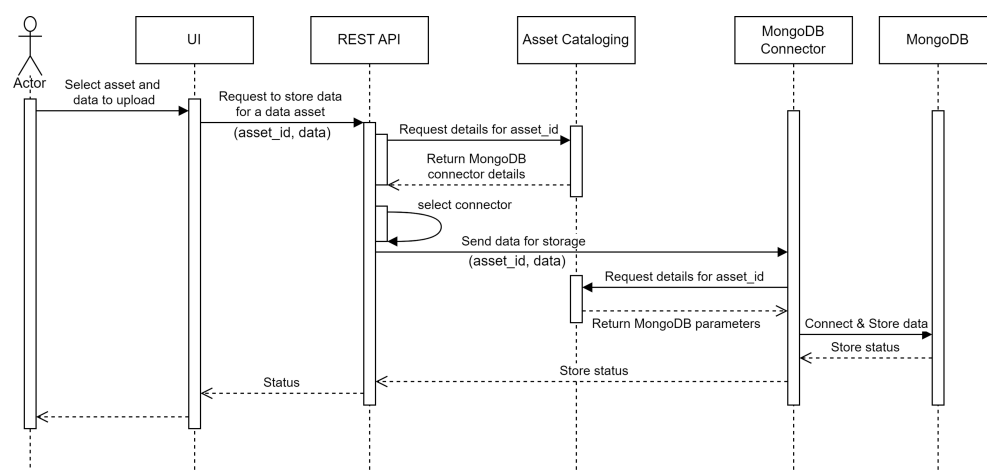


Figure 6. UML sequence diagram illustrating the data upload path. A user request enters the Backend, which queries Asset Cataloging for the target asset metadata, selects and invokes the appropriate connector, and writes the payload to the target repository.

Following a single round-trip makes the pattern concrete. On the write side (Figure 8, the producer/upload path), the platform Backend, part of the Big Data Platform (BDP) service and labeled BDP::Data manipulation in the diagram, requests connector details from Flow Management (BDP::Management), which resolves them from Asset Cataloging. The producer sets up the Kafka client using the returned parameters and publishes data to the Kafka topic. Flow Management then initializes the Flow Engine, which subscribes to the topic, receives the streamed data, and routes them through the Connector Layer to Data Storage. The read side (Figure 9, the consumer/retrieval path) mirrors this sequence: the Backend again resolves connector details from Asset Cataloging via Flow Management and sets up the Kafka client, but here, it subscribes to the topic and streamed data flows directly back to the consumer. The two directions differ only in their final hop (into storage versus back to the caller) while sharing the identical metadata-resolution prefix.

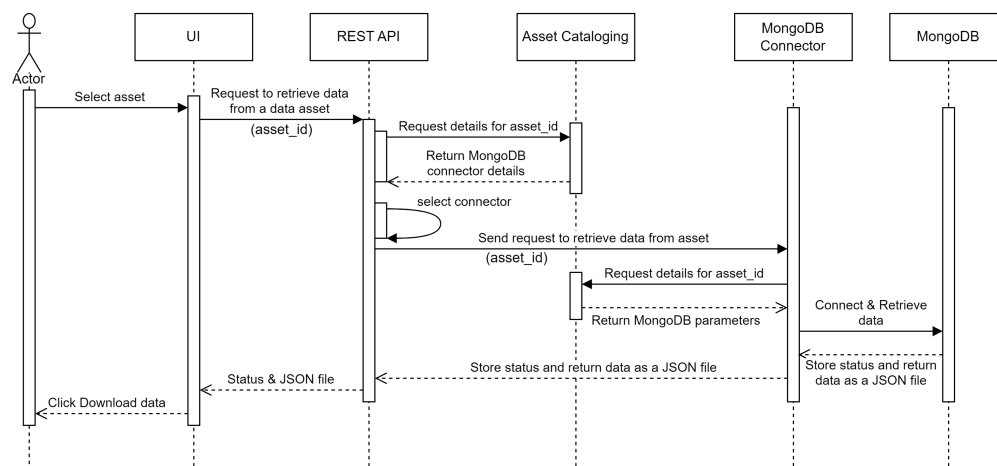


Figure 7. UML sequence diagram showing data retrieval flow: User request reaches the Backend, which queries Asset Cataloging for connector metadata, selects and invokes the appropriate connector, retrieves data from the target repository, and returns them to the caller in their original format.

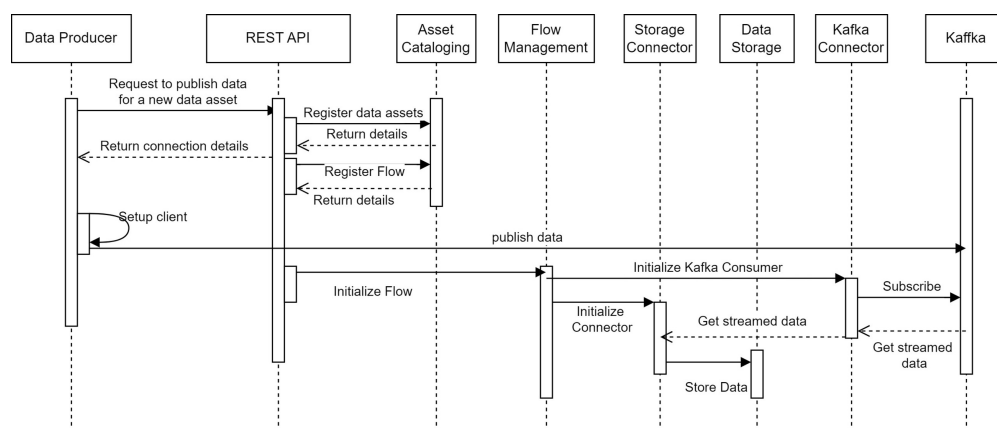


Figure 8. Sequence diagram of the data upload flow with explicit Kafka interaction, from producer publication through the BDP Backend and the Asset Cataloging lookup to Flow Engine routing through the Connector Layer into storage.

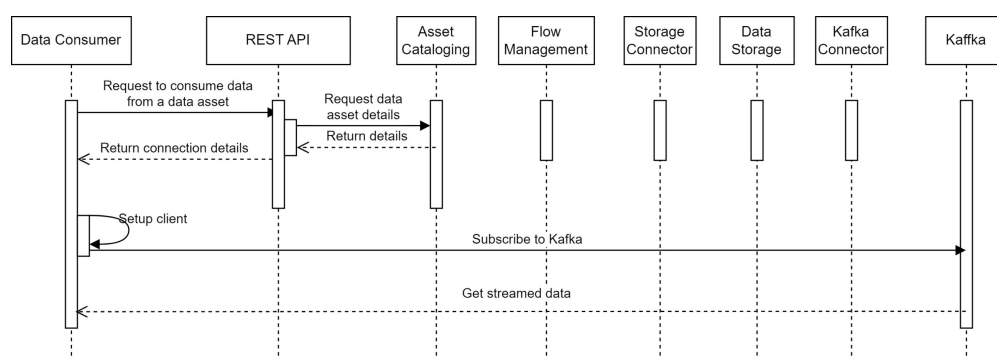


Figure 9. Sequence diagram of the data consumption flow with explicit Kafka interaction, from consumer request through the BDP Backend and the Asset Cataloging lookup to direct Kafka streaming back to the consumer.

4.4.2. New-Connector Registration

Adding a new storage technology requires no changes to the platform core. A new connector is registered by POSTing a connector document—naming its four endpoint operations and their URLs—to `POST /cataloguing/connector`, with the following form:

```
{
```

```

"connector": "<connector-name>",
"description": "<human-readable description>",
"endpoints": [
  { "name": "retrieve", "method": "GET", "url": ".../api/data/retrieve" },
  { "name": "file_retrieve", "method": "GET", "url": ".../api/file/retrieve" },
  { "name": "store", "method": "POST", "url": ".../api/data/upload" },
  { "name": "file_upload", "method": "POST", "url": ".../api/file/upload" }
]
}

```

The URLs point to the connector microservice. The storage backend connection details—server address, credentials, database, collection or topic—are stored separately in the asset's `interface.parameters` and passed to the connector at request time. This separation allows multiple assets to share the same connector document while pointing to different storage instances; for example, two MongoDB assets for different pilot sites reference the same MongoDB connector but carry distinct `mongoURI`, `database`, and `collection` values in their parameters. Once registered, any asset can reference the connector by its document ID in `interface.connector_id`, making the new storage technology immediately accessible without redeploying the platform.

4.5. Integration Dimensions

Following RAMI 4.0 terminology [51], the platform addresses three integration dimensions critical for Industry 4.0 and IoT environments:

- (i) **Vertical Integration** (Digital ↔ Physical): Sensor data flow from physical IoT devices through the layered architecture to digital storage repositories. Asset metadata capture device specifications at the perception layer, the communication layer handles protocol translation, and connectors bridge to databases at the enterprise layer. This vertical flow supports closed-loop control; analytics results can flow back to actuators.
- (ii) **Horizontal Integration** (Cross-System): Flow Management orchestrates data movement across heterogeneous repositories belonging to different organizational systems or pilot sites. A flow transferring data from one pilot's MongoDB to another pilot's external enterprise system exemplifies horizontal integration. The platform mediates cross-system data exchange without requiring direct communication between systems.
- (iii) **End-to-End Integration** (Physical Infrastructure Preservation): Unlike platforms requiring migration to unified storage, our approach links physical IoT devices to diverse digital repositories without replacing existing infrastructure. Metadata specifications provide the semantic bridge. Assets describe “what” data mean independent of “how” they are stored, spanning the full production lifecycle while respecting organizational technology investments.

4.6. Usage Scenarios

Data producers (sensors, gateways, services) generate data flowing to consumers (storage systems, analytics applications), as illustrated in Figure 10. Systems may embody both roles; a predictive maintenance service consumes sensor data while generating maintenance predictions.

Consider a temperature sensor generating readings at 10 ms intervals. An edge gateway aggregates and publishes batches to Kafka. The platform queries Asset Cataloging, discovers the configured MongoDB destination, and creates a data flow automatically. The MongoDB connector retrieves from Kafka and stores data in their original format. An analytics application follows the same pattern: query the registry, discover the temperature asset, retrieve historical data.

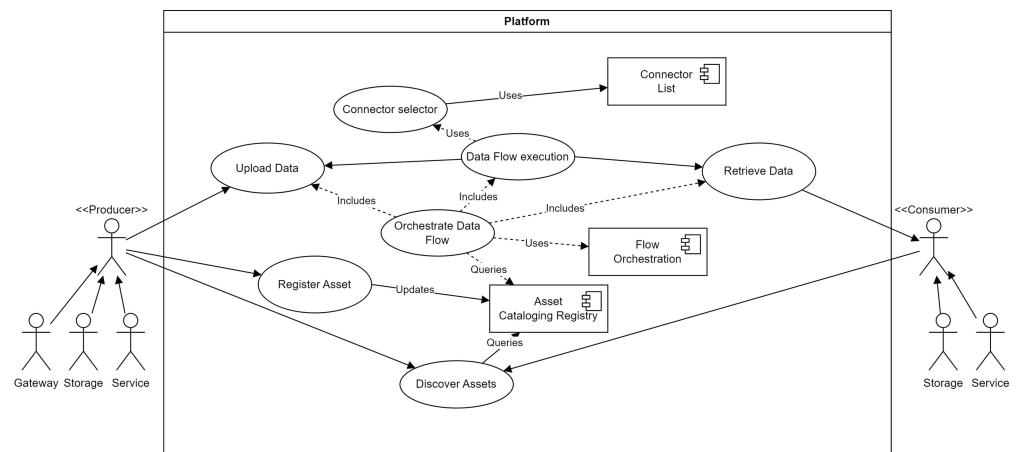


Figure 10. Usage scenario of the platform in practice: data producers (sensors, gateways) and consumers (storage, analytics) interact through the five-layer architecture, illustrating upload, retrieve, and flow operations. The predictive maintenance service exemplifies bidirectional flows.

5. Implementation

We implemented the platform using a technology stack combining Node.js with MongoDB/Kafka clients and containerized deployment via Docker.

5.1. Development Methodology and Testing

Implementation followed systematic verification: unit testing per microservice, integration testing for inter-service communication (particularly complex multi-connector flows), and system testing for the web interface. Swagger generated API documentation automatically.

5.2. Technology Stack

The platform was built on Node.js microservices with MongoDB and Kafka clients and a browser-based web frontend. The Hadoop HDFS connector used the WebHDFS REST interface, mapping the standard four operations to WebHDFS REST calls—file retrieval via OPEN, file upload via CREATE, and directory enumeration via LISTSTATUS—providing large-scale distributed file storage access without requiring MapReduce or data consolidation at pilot sites. Docker and Docker Compose supported containerized deployment, and Swagger provided automated API documentation.

The five logical layers described in Section 4 mapped to five microservices in the implementation: Backend (implementing Communication and Interface layers), Asset Cataloging, Flow Management, Connectors, and Web UI. Backend's primary REST API is detailed in Table 7.

Table 7. Platform Backend API exposing high-level operations for end users and external systems. These endpoints orchestrate metadata queries and connector invocations.

Endpoint	Description
GET /api/asset-cataloguing	List all registered assets
GET /api/flows	List all data flows
POST /api/data-flow	Create data flow between two assets
POST /api/stop-data-flow/{id}	Stop active flow
DELETE /api/data-flow/{id}	Delete flow
GET /api/data-retrieve/{asset_id}	Retrieve data from asset repository
POST /api/data-upload/{asset_id}	Upload data to asset repository

The Kafka connector commits consumer offsets only after the MongoDB write acknowledgment (or REST API response) is received, implementing at-least-once delivery semantics: a failed storage operation leaves the offset uncommitted, triggering automatic reprocessing without data loss.

5.3. Service-Oriented Architecture Implementation

Following the SOA principles outlined in Section 4, we implemented the platform as independent microservices communicating through well-defined REST APIs and message queues. This service-based implementation supports the connector pattern extensibility described architecturally; new connectors integrate as independent services without modifying the platform core.

5.4. API Implementation

The three-tier REST API architecture (Platform API, Cataloging API, Connector APIs) described in Section 4 was implemented using Node.js with Express framework. The Backend service exposes the Platform API endpoints shown in Table 7, orchestrating metadata queries and connector invocations to fulfill user requests. Each service implements its API tier independently, enabling parallel development and deployment.

5.5. Security Implementation

The platform realizes the security and governance requirements of Section 3.5 through enforcement points distributed across the deployment stack rather than a dedicated security service. All inter-service communication uses TLS; because streaming payloads are batch-aggregated (Section 6), negotiation applies once per batch rather than per sample, keeping overhead negligible. Role-based access control is enforced at three levels—asset access, connector invocation, and flow management—with Web UI authentication delegated to Keycloak, so the platform stores no user credentials. Authorization is metadata-driven: asset ownership, pilot-site assignment, and data-classification attributes act as executable access policies, so a user scoped to one pilot cannot create flows over another pilot's assets without hardcoded permission lists. Every integration operation produces an append-only audit record supporting post hoc forensic analysis. Consistent with the threat model of Section 3.5, the platform assumes a trusted enterprise network and delegates perimeter defense to existing firewall and intrusion-detection infrastructure; adversarial security benchmarking and SIEM integration are deferred to future work (Section 8.1).

5.6. Deployment Architecture

Each service runs in an isolated Docker container; Docker Compose defines configurations, network connectivity, and dependencies across the full stack (Figure 11).

Services communicate over a private network, with only the Backend API and Web UI exposing external access while internal services remain isolated; persistent volumes guarantee data survive container restarts. This packaging enabled rapid deployment to pilot sites requiring only Docker and the compose file and delivers the operational benefits of a containerized architecture: independent service scaling, zero-downtime rolling updates, environment parity, and portability across on-premise or cloud environments.

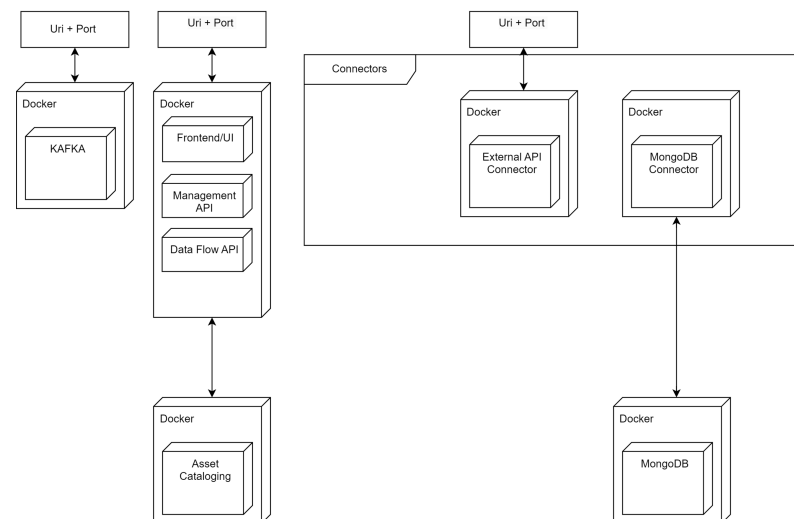


Figure 11. Containerized deployment architecture. Each service (Frontend, Backend, Asset Cataloging, Connectors, Kafka) runs in isolated Docker containers. Services communicate through private Docker network. Only Backend API (port 8080) and Web UI (port 3000) expose external access. MongoDB provides persistent storage for Asset Cataloging metadata and optionally pilot data.

6. Evaluation and Validation

We validated the platform through three scenarios using real mining sensor data, demonstrating industrial-scale data handling with heterogeneous formats across multiple repositories while maintaining integrity.

6.1. Evaluation Methodology

Evaluation targeted six critical requirements from Section 3: heavy data streaming, heterogeneous format handling, multiple repository management, historical batch support, real-time streaming, bidirectional operations (retrieve, upload, flow).

We utilized a real-world dataset from a MINE.IO Research and Teaching Mine “FLB Reiche Zeche” pilot site at TU Bergakademie Freiberg containing 137.2 million sensor readings captured during actual mining operations. The dataset totaled 13.98 GB; sensor sampling rates reached up to 10,000 Hz, with edge computing grouping 1000 readings into UDP packets transmitted at 10 Hz (every 100 ms). Data represented diverse sensor types including temperature, pressure, vibration, length, rotational speed and environmental monitors. Data formats included structured JSON, CSV, and compressed Parquet files.

We deployed the test environment on a virtual machine (eight vCPUs AMD EPYC-Rome, 16 GB of RAM, 74 GB SSD, local Docker network). All platform services, the Kafka broker, and the MongoDB instance ran as Docker containers on this single host, eliminating inter-host network transmission time and ensuring consistent I/O performance across evaluation scenarios. Services ran without explicit container resource limits; Kafka was configured with a 10 GB maximum partition size and 1-day log retention. The test environment integrated three data source types: Apache Kafka (real time streaming), MongoDB (document storage), and external REST APIs (enterprise document management). This multi-source setup validated the platform’s heterogeneous integration capabilities.

Latency was measured using application layer timestamps recorded in Node.js via `Date.now()` at three points: event creation (Kafka message production), event reception (connector consumer), and storage completion (MongoDB write acknowledgment). Because all components co-resided on the same virtual machine, application layer timestamps were sufficient; no inter-host clock synchronization (e.g., NTP/PTP) was required. End-to-end latency was defined as the interval from event creation to storage completion.

Reported latencies therefore reflect pure processing and I/O overhead rather than inter-host communication. `Date.now()` had approximately 1 ms resolution; for the 5–40 ms steady-state range, this introduced up to 20% uncertainty on individual measurements, though aggregate statistics across thousands of events remained largely unaffected by this variance.

Three scenarios validated different platform capabilities: Scenario A tested real-time Kafka-to-MongoDB data flows with format diversity; Scenario B validated large-scale file retrieval; Scenario C tested batch historical data upload. All scenarios used production-grade data volumes for realistic validation.

6.2. Test Data Preparation and Streaming Simulation

To simulate realistic IoT sensor deployments, a replay pipeline—a Node-RED edge platform driven by a custom `parser.js` script—streamed the full dataset over UDP and fed three parallel Node-RED flows, each writing to a distinct Kafka topic with a different fidelity/format tradeoff: *sensorial-test* (full-fidelity JSON preserving all 1000 readings per packet; 137,200 messages), *short-test* (aggregated JSON, per-variable averages; 137,200 messages), and *parquet-test* (60 s buffered, compressed Parquet for archival; 229 messages). Edge counters tracked message production for later verification against stored counts. This multi-format approach exercised the platform’s heterogeneous data handling, with JSON documents, aggregated summaries, and compressed binary files flowing simultaneously through different processing paths.

The `parser.js` Node.js script read the full dataset and streamed it via UDP packets to simulate real-time sensor transmission. Each UDP packet contained 1000 readings (100 ms of sensor data at 10,000 Hz sampling rate), transmitted every 100 ms to replicate the 10 Hz edge transmission rate. Node-RED flows received the UDP packets and implemented three processing strategies:

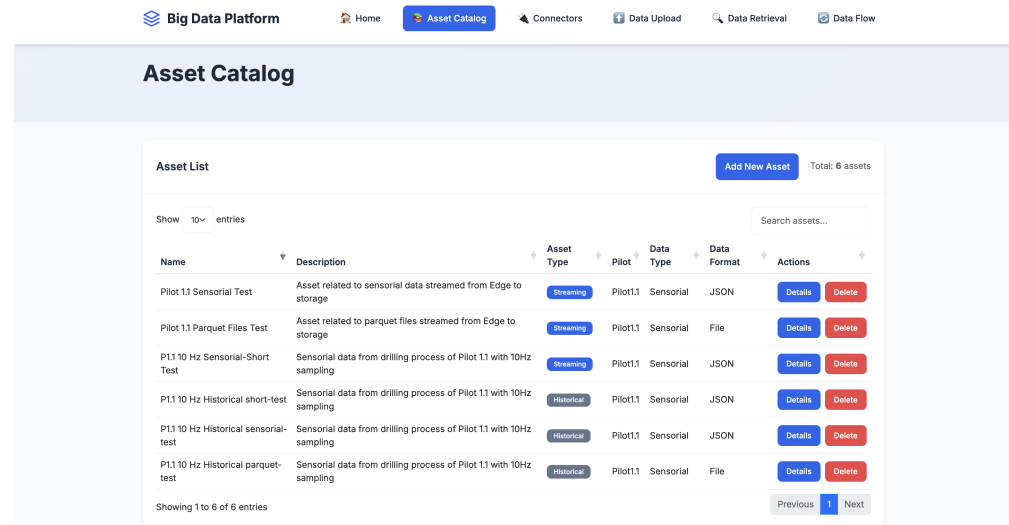
1. **Detailed JSON storage** (*sensorial-test* topic): Each UDP packet converted to a JSON object containing arrays for each sensor variable, preserving all 1000 readings per packet. This represents full-fidelity historical data storage. The flow produced 137,200 Kafka messages.
2. **Aggregated JSON storage** (*short-test* topic): Each UDP packet converted to a JSON object containing only average values per sensor variable, reducing 1000 readings to single aggregate values. This represents summary statistics for dashboards or alerts. The flow produced 137,200 Kafka messages.
3. **Compressed file storage** (*parquet-test* topic): The system buffered UDP packets until 600 packets (≈ 60 s of data) were accumulated, then aggregates them into CSV, converted to compressed Parquet, and sent the result as a single Kafka message. This flow represents archival storage optimized for compression efficiency; the 60 s buffering window deliberately targeted long-term archival rather than real-time monitoring. The flow produced 229 Kafka messages. To accommodate the compressed Parquet payloads, the Kafka broker `message.max.bytes` parameter was set to 5 MB; the resulting messages averaged approximately 3 MB, well within the configured limit.

Edge counters tracked message production, enabling verification against subsequent storage counts.

6.3. Scenario A: Real-Time Data Flow Integration

Scenario A validated consuming high-volume Kafka streams and routing to MongoDB storage based on data format, testing core integration capabilities. Three parallel flows transferred data from Kafka topics to MongoDB collections: *sensorial-test* (JSON arrays),

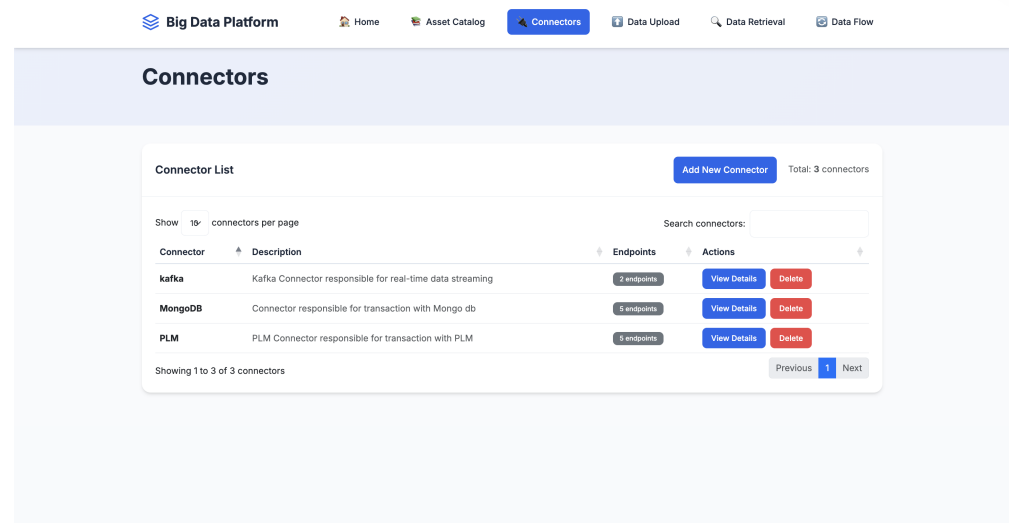
short-test (aggregated JSON), and parquet-test (binary files). Assets and connectors were registered in Asset Cataloging with appropriate metadata (Figures 12 and 13).



The screenshot shows the 'Asset Catalog' page of the 'Big Data Platform'. It features a navigation bar with links to Home, Asset Catalog, Connectors, Data Upload, Data Retrieval, and Data Flow. The main content area is titled 'Asset Catalog' and contains an 'Asset List' table. The table has columns for Name, Description, Asset Type, Pilot, Data Type, Data Format, and Actions. There are six rows of assets, each with a 'Details' and 'Delete' button. A search bar and pagination controls are also visible.

Name	Description	Asset Type	Pilot	Data Type	Data Format	Actions
Pilot 1.1 Sensorial Test	Asset related to sensorial data streamed from Edge to storage	Streaming	Pilot1.1	Sensorial	JSON	Details Delete
Pilot 1.1 Parquet Files Test	Asset related to parquet files streamed from Edge to storage	Streaming	Pilot1.1	Sensorial	File	Details Delete
P1.1 10 Hz Sensorial-Short Test	Sensorial data from drilling process of Pilot 1.1 with 10Hz sampling	Streaming	Pilot1.1	Sensorial	JSON	Details Delete
P1.1 10 Hz Historical short-test	Sensorial data from drilling process of Pilot 1.1 with 10Hz sampling	Historical	Pilot1.1	Sensorial	JSON	Details Delete
P1.1 10 Hz Historical sensorial-test	Sensorial data from drilling process of Pilot 1.1 with 10Hz sampling	Historical	Pilot1.1	Sensorial	JSON	Details Delete
P1.1 10 Hz Historical parquet-test	Sensorial data from drilling process of Pilot 1.1 with 10Hz sampling	Historical	Pilot1.1	Sensorial	File	Details Delete

Figure 12. Asset Cataloging registry showing six registered assets: three streaming assets (Kafka topics for sensorial-test, short-test, parquet-test) and three historical storage assets (corresponding MongoDB collections). Each asset contained metadata specifying connector type, data format, and connection parameters.



The screenshot shows the 'Connectors' page of the 'Big Data Platform'. It features a navigation bar with links to Home, Asset Catalog, Connectors, Data Upload, Data Retrieval, and Data Flow. The main content area is titled 'Connectors' and contains a 'Connector List' table. The table has columns for Connector, Description, Endpoints, and Actions. There are three rows of connectors, each with a 'View Details' and 'Delete' button. A search bar and pagination controls are also visible.

Connector	Description	Endpoints	Actions
kafka	Kafka Connector responsible for real-time data streaming	2 endpoints	View Details Delete
MongoDB	Connector responsible for transaction with Mongo db	5 endpoints	View Details Delete
PLM	PLM Connector responsible for transaction with PLM	5 endpoints	View Details Delete

Figure 13. Connector Cataloging showing the three connectors used in the controlled evaluation (Kafka, MongoDB, External API). Each connector implemented the standard four-operation interface (retrieve, file_retrieve, store, file_upload) with technology-specific URLs and connection logic.

Flows were activated via the web interface, creating continuous data transfer processes. The Kafka connector consumed messages from topics while the MongoDB connector stored them in corresponding collections. We monitored Kafka using Offset Explorer to track message availability and MongoDB using Compass to verify storage.

All three flows achieved 100% transfer accuracy (Table 8): sensorial-test and short-test each delivered 137,200 messages to MongoDB without loss, while parquet-test delivered all 229 Parquet files. MongoDB collections showed exact correspondence with Kafka topic message counts, validated through document count queries.

Table 8. Scenario A validation results. Three heterogeneous flows ran simultaneously under metadata-driven routing; format integrity confirmed correct connector selection and payload handling for each type.

Flow	Messages	Data Format	Stored	Format Verification
sensorial-test	137,200	JSON (arrays)	137,200	All 1000 readings/packet preserved per document
short-test	137,200	JSON (aggregated)	137,200	Aggregate field structure intact
parquet-test	229	Parquet (binary)	229	Bit-exact reconstruction verified (MD5)

Latency across 5000 events (batch size 1) is captured in Figure 14. Initial events show elevated latency due to system initialization (connection pooling, buffer warming). After warm-up, steady-state latency (events 2001–5000) reaches a mean of 10.1 ms (± 4.0 ms std, P99 26 ms, max 35 ms), demonstrating performance suitable for industrial applications.

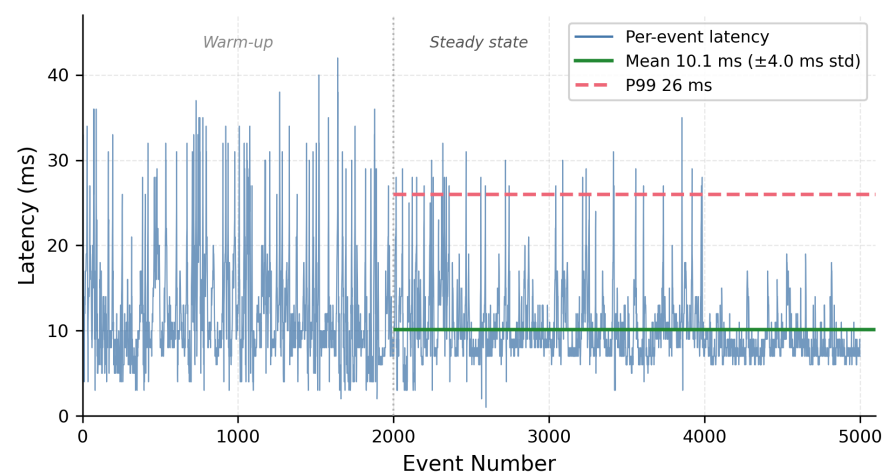


Figure 14. Latency from event creation to MongoDB storage across 5000 events (batch size 1). Steady-state (events 2001–5000): mean 10.1 ms (± 4.0 ms std, P99 26 ms). Initial elevated latency reflects connection pooling and buffer warming.

Scenario A confirmed heterogeneous format handling across JSON arrays, aggregated JSON, and binary Parquet; simultaneous multi-flow operation without interference; and real-time streaming sustained over hours.

6.4. Scenario B: Large-Scale File Retrieval

Scenario B validated reverse data flow by retrieving 229 binary Parquet files from MongoDB. The MongoDB connector reconstructed files from Base64 encoding, aggregated them into a ZIP archive, and returned them for download. MD5 hash comparison verified bit-exact preservation, validating format integrity through the complete storage and retrieval cycle (Table 9).

Table 9. Scenario B validation results. Base64-to-binary reconstruction and ZIP aggregation confirm that binary format integrity is preserved through the complete MongoDB storage and retrieval cycle.

Stage	Count	Format	Integrity Check
Stored in MongoDB	229	Base64-encoded Parquet	–
Retrieved as archive	229	ZIP (reconstructed Parquet)	MD5 match: bit-exact reconstruction confirmed

Scenario B thus validated large-scale reverse data flow—229 binary files retrieved in a single operation and ZIP-aggregated—with bit-exact format preservation end to end.

6.5. Scenario C: Historical Batch Data Upload

Scenario C validated batch data upload by retrieving 137,200 JSON documents from MongoDB, then uploading them back to the same collection, effectively doubling stored data. This round-trip operation validated idempotent operations and format preservation. Document count increased from 137,200 to exactly 274,400, with MongoDB Compass queries verifying structural integrity (Table 10).

Table 10. Scenario C validation results. Round-trip MongoDB → JSON export → MongoDB upload confirms structural preservation and correct idempotent batch insertion.

Stage	Documents	Verification
Source (MongoDB)	137,200	–
After round-trip upload	274,400	Field names, types, and nesting preserved; exact doubling confirms zero loss and correct idempotent insertion

Scenario C thus validated high-throughput batch upload of large JSON exports with memory-efficient processing and round-trip format preservation across the full MongoDB-to-JSON-to-MongoDB cycle.

6.6. Performance Analysis

Beyond functional validation, we analyzed performance characteristics. Payload format dominated latency, as confirmed by Figure 15. Compressed binary Parquet achieved sub-100 ms average, while JSON formats were an order of magnitude slower due to payload size. Compact aggregated JSON reduced mean latency by approximately 35% relative to full JSON arrays. Connector dispatch fell below the 1 ms measurement resolution for all connector types, confirming that all measured latency was attributable to MongoDB write I/O rather than platform orchestration cost.

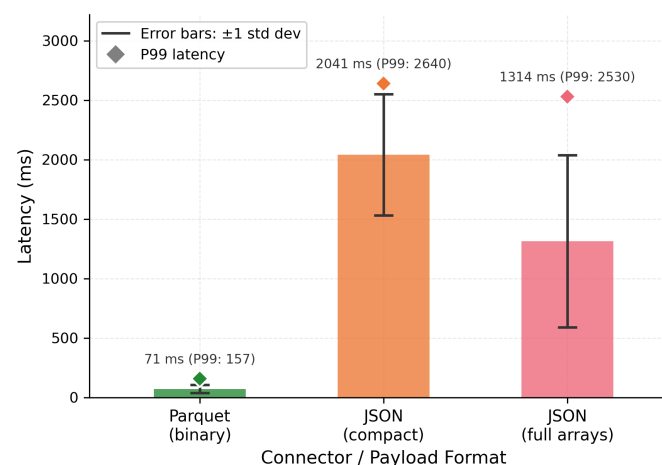


Figure 15. Latency distribution across connector implementations and data formats. MongoDB-Parquet-file: avg 71 ms (± 35 ms std, P99 157 ms). MongoDB-JSON-compact: avg 1314 ms (± 725 ms std, P99 2530 ms). MongoDB-JSON: avg 2040 ms (± 509 ms std, P99 2640 ms).

Figure 16 examines throughput scalability under varying concurrent flow loads. Throughput scales with concurrent flows up to a peak at 10 flows, after which degradation reflects a shared MongoDB write bottleneck rather than platform architectural

limitations. Docker container monitoring verified that Backend and Flow Management services maintained stable, low CPU and memory utilization throughout the degradation period; the bottleneck is therefore external to the orchestration layer and attributable to MongoDB I/O saturation under concurrent write pressure. Linear scalability up to 10 flows indicates efficient resource utilization within the platform itself.

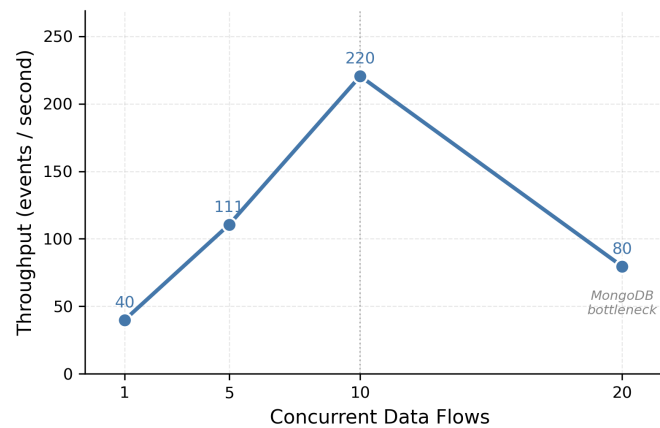


Figure 16. Platform throughput under varying concurrent flow loads. Throughput scales from 1 to 10 concurrent flows, reaching peak performance at 10 flows with degradation at 20 flows reflecting shared MongoDB bottleneck rather than platform architecture limitations.

Concurrency effects on per event latency (JSON connector, batch = 100) are detailed in Table 11, complementing the throughput view in Figure 16. Mean latency grows from 1314 ms at one flow to 5195 ms at twenty flows; the platform orchestration layer remained stable throughout (confirmed by Docker monitoring), so the growth reflects shared MongoDB write queue pressure rather than architectural limits. Deployments requiring sub-second-per-event latency should either limit concurrent flows or scale the storage backend horizontally.

Table 11. Per-event latency under increasing concurrent flow load (JSON connector, batch = 100). Latency grows with concurrency as MongoDB write contention increases; the platform orchestration layer remained stable throughout, confirmed by Docker monitoring of Backend and Flow Management services.

Concurrent Flows	Mean Latency	Std	P99
1	1314 ms	±725 ms	2530 ms
5	2022 ms	±1502 ms	11,902 ms
10	3308 ms	±2134 ms	11,738 ms
20	5195 ms	±3966 ms	14,139 ms

6.7. Performance Characteristics

Performance testing validated the platform's ability to sustain industrial IoT workloads. Through production-ready optimizations including batch aggregation, connection pooling, backpressure control, and flow management, the platform achieved sustained processing of message streams representing 10,000 Hz sensor sampling data with no observed message loss across the complete 137.2 million record dataset. Key performance metrics from validation are summarized in Table 12.

The batch size configures the latency/throughput tradeoff. With batch = 1, mean latency was 10.1 ms (±4.0 ms std), suitable for per-event real-time streaming. With batch = 100, mean latency was 533.6 ms (±299.7 ms std, P99 1042 ms) through amortized insertion cost, remaining under 1 s P99 for archival workloads.

The platform implements at-least-once delivery semantics. Automatic retry with exponential backoff (three attempts) recovers failed connector operations, and idempotent storage operations (MongoDB insertions keyed on unique message identifiers) prevent duplicate records, achieving zero net data loss in practice. We adopted at-least-once delivery deliberately rather than as a limitation. Exactly once delivery would require distributed transaction coordination spanning the streaming layer and each storage backend (for example, Kafka transactions with a two-phase commit into MongoDB), whose coordination latency is incompatible with the 10,000 Hz ingestion rates targeted in Section 3. Out-of-order delivery within a topic is prevented by Kafka partition ordering; cross-partition ordering, where required, is resolved at the application layer using message timestamps. In combination, at-least-once delivery and idempotent, identifier-keyed writes yield the same end state as exactly once for the mining integration workload—exact source-to-destination record counts and zero net loss across 137.2 million records (Table 13)—without the throughput penalty of distributed commit. Applications requiring transactional exactly once semantics can layer them above the connector interface without changing the platform.

Table 12. Platform performance characteristics achieved during validation with 137.2M real mining sensor records. System sustained 10,000 Hz sensor sampling (10 Hz edge transmission) while supporting concurrent batch operations across multiple repositories.

Performance Metric	Achieved Value
Sensor Sampling Rate	up to 10,000 Hz (1000 readings per 100 ms packet)
Total Records Processed	137.2 million
Dataset Size	13.98 GB
Concurrent Flows	3 simultaneous (Kafka → MongoDB)
Data Loss	0% (100% accuracy)
Memory Footprint	<2 GB (observed via Docker container monitoring)
Database Operations	Batch insertion (100 records/operation)
Connection Management	Persistent pool (5 connections)

Table 13 maps each evaluation scenario to the six requirements from Section 3. Source-to-destination record counts matched exactly in all scenarios, format integrity was verified per scenario, and zero message loss was observed across 137.2 million records spanning four pilot sites.

Table 13. Validation summary mapping evaluation scenarios to the six system requirements.

Requirement	Validation Evidence
Heavy Streaming	Processed 137.2M records (13.98 GB); sensors at up to 10,000 Hz, edge transmission at 10 Hz
Heterogeneous Data	Handled JSON, CSV, Parquet formats simultaneously
Multiple Repositories	Integrated Kafka, MongoDB, external APIs concurrently
Historical Data	Batch upload/retrieval of 137.2K+ records
Real-Time Streaming	Sustained 10 Hz edge transmission (10,000 Hz sampling) without data loss
Data Operations	Validated retrieval, upload, automated workflows

6.8. Operational Deployment

Following evaluation, the platform has been operational for eight months across four pilot sites spanning diverse mining and marine scenarios and heterogeneous storage infrastructures. Pilot 1 monitors active mine drilling operations; 12 drill-hole experiments

were executed during that period, each lasting approximately two days, with real-time Kafka streaming producing 10–15 GB per experiment (~140 GB accumulated) stored in MongoDB (240 GB NVMe, four vCPUs, 16 GB of RAM). Pilot 2 processes 3D point cloud data from muon telescope observations; three campaigns of three months each produced approximately 5 GB per campaign (15 GB total) stored in MongoDB (74 GB NVMe, four vCPUs, 8 GB of RAM) via batch ingestion due to limited edge connectivity. Pilot 3 ingests seismic and electrical resistance sensor streams over one-month observation periods; one month of preprocessed sensor data was stored (approximately 200 MB) on the same MongoDB infrastructure as Pilot 2, with data preprocessing performed externally and real-time streaming used for ingestion. Pilot 4 integrates an underwater robot and muon telescope for subsurface exploration; one exhibition campaign of 1–2 days transmitted data as continuous chunks to a Hadoop HDFS data lake (four data nodes, four vCPUs each, ~280 GB total capacity). Across all deployments, no critical failures or data loss events were recorded; minor network interruptions were automatically recovered via the platform's retry mechanism and Kafka consumer group offset management. Collectively, the four pilots span a distinct data type each (high-frequency time-series, binary 3D point clouds, structured geophysical measurements, and binary robotic telemetry), two storage backends (MongoDB and Hadoop HDFS), two ingestion patterns (Kafka streaming and batch file upload), and two industrial domains (mining and marine). The consistency of zero message loss and sub-second connector dispatch latency across all four, despite their diversity, is direct empirical evidence that the metadata-driven architecture generalizes across heterogeneous industrial data sources without domain-specific orchestration logic.

7. Discussion

This section interprets evaluation findings, positions our metadata-driven approach relative to existing platforms, and examines applicability beyond the mining domain.

7.1. Summary of Evaluation Findings

Scenarios A, B, and C confirm that metadata-driven orchestration routes heterogeneous flows at runtime without hardcoded logic, that the connector abstraction handles diverse formats (JSON arrays, aggregated JSON, binary Parquet) with negligible overhead, and that the unified architecture sustains both streaming and batch operations at industrial scale without message loss. Unlike dual-lane approaches such as FASTEN [17] that use metadata for lane selection, our platform uses metadata as complete integration specifications, driving connector selection and invocation at runtime without workflow coding.

Steady-state performance requires initialization (connection pooling, buffer warming), so the approach suits long-running integration processes more than ephemeral operations. The platform targets near-real-time streaming (the mean 10.1 ms end-to-end latency of Scenario A comfortably meets the sub-second responsiveness required in Section 3) rather than hard real-time guarantees with deterministic millisecond deadlines. Performance degrades beyond 10 concurrent flows as MongoDB write contention accumulates; Docker monitoring confirmed the orchestration layer stayed stable throughout, placing the bottleneck in the storage backend rather than the platform. Sites requiring higher throughput should therefore scale the storage backend horizontally rather than limit concurrency, using either of two standard MongoDB mechanisms. Sharding distributes writes across nodes by partitioning on a shard key: for high-volume sensor collections, a hashed key on a high-cardinality field such as the device identifier (`{deviceId: "hashed"}`) spreads inserts evenly and avoids the write hotspot a monotonic timestamp key would create. Replica sets, alternatively, distribute reads across replicas while a primary handles writes. Both are transparent to the platform: because the orchestration layer is stateless and storage-

agnostic, scaling the backend requires no core changes—only updating the `mongoURI` in the relevant asset's `interface.parameters` to the scaled cluster endpoint, after which the existing metadata-driven invocation path routes all operations to it automatically.

The security and governance requirements (RBAC, TLS, audit logging) were exercised across pilot deployments but were not the focus of this evaluation; Web UI authentication was delegated to Keycloak (Section 5.5). Batch aggregation mitigates TLS overhead on high-frequency streams: at 10,000 Hz, readings are aggregated into Kafka messages of 1000 before transmission, so TLS applies once per batch rather than per sample. The per-batch overhead (typically under 1 ms) is negligible against the mean 10.1 ms (P99 26 ms) steady-state latency of Scenario A. We report no raw throughput comparison against general-purpose middleware (Apache NiFi, Kafka Connect, Apache Flink). A naive comparison would confound two distinct quantities: the platforms' intrinsic capability and the effort of hand-authoring the flow graphs, connectors, or streaming jobs each requires—the manual integration work our approach is designed to eliminate. A fair benchmark must instead isolate the storage-write path, holding sink, payload, and hardware identical so that the measured difference reflects orchestration overhead alone; we scope this controlled comparison as the primary open evaluation gap (Section 8). The comparison we can make rigorously here is architectural: Table 1 spans the dimensions relevant to IIoT integration (infrastructure preservation, workflow coding, metadata executability, and backend extensibility), showing where the platform offers advantages and where established tools remain stronger.

7.2. Generalizability Beyond Mining

Developed for mining, the platform's design principles are likely applicable to any domain with heterogeneous IoT data across multiple repositories. Smart cities, healthcare, and manufacturing share the same pattern: diverse sensor types, multiple storage silos, and a requirement to preserve existing infrastructure [10,52–54]. Domain adaptation requires implementing domain-specific connectors and extending metadata schemas with domain fields; the core architecture needs no modification. The Hadoop HDFS connector (already implemented for Pilot 4 via WebHDFS) demonstrates this extensibility: integrating a distributed file storage backend required no changes to the platform core, only connector registration in Asset Cataloging. Further examples include InfluxDB for time series data or DICOM for medical imaging.

The platform operates at the technical integration layer and is by design complementary to, not competitive with, established IoT semantic frameworks. FIWARE addresses semantic standardization: it defines how data are modeled (NGSI-LD entities, Smart Data Models) and provides a Context Broker for managing live context information through a standardized API. The proposed platform addresses a different problem: how data move across heterogeneous repositories without replacing existing infrastructure. These concerns are architecturally distinct and naturally composable.

Integration is also edge-agnostic, beginning at the registered asset endpoint rather than in the edge tier. It assumes edge devices or gateways perform initial aggregation and protocol translation (as in our evaluation, where a Node-RED gateway packetizes and batches raw UDP sensor readings before publishing to Kafka), keeping the platform decoupled from edge-specific protocols, consistent with standard IIoT practice. To validate this concretely, we implemented a pilot FIWARE integration: a FIWARE IoT agent transformed edge telemetry into NGSI-LD format, and the FIWARE Context Broker published these standardized entities to a Kafka topic. From the platform's perspective that topic was simply a data asset—the registry stored its bootstrap-server configuration and topic identifier, and flow orchestration consumed and persisted the stream to a target

repository—requiring no knowledge of the upstream FIWARE deployment. The same logic applies to any other edge framework (W3C WoT gateways, custom IoT processors) that exposes a supported endpoint.

W3C WoT Thing Descriptions describe device affordances and protocol bindings that align naturally with the asset's `interface_*` connection parameters and could populate asset metadata automatically upon device registration, without the platform needing to implement WoT semantics internally. At the metadata model level, the asset corresponds to `dcat:Dataset` and the connector to `dcat:DataService`; flow represents an execution-layer extension with no DCAT equivalent, reflecting the platform's role beyond discovery into runtime orchestration.

This architectural stability aligns with reference architecture principles for future IIoT networks [42], which emphasize modularity and adaptability to cross-organizational scenarios.

7.3. Design Implications of Metadata-Driven Integration

The metadata-driven approach succeeds when three conditions hold: *integration complexity* exceeds data processing complexity (heterogeneous sources with simple transformations), *infrastructure preservation* is economically or operationally mandated, and *metadata completeness* is achievable (source characteristics are knowable and expressible). Conversely, it proves less suitable when processing logic dominates (complex transformations, machine learning pipelines), when greenfield deployments permit unified storage, or when metadata are inherently incomplete or unstable (undocumented proprietary formats). These conditions held in the MINE.IO context and define the boundary under which these findings generalize.

8. Conclusions

In this work, we presented a metadata-driven execution model for the unified integration and management of heterogeneous IoT data sources, in which integration behavior is resolved at runtime from executable metadata rather than encoded in static workflows. Our evaluation shows that executable metadata can drive heterogeneous IoT integration without workflow reconfiguration or infrastructure replacement: the platform selects and invokes the appropriate connector at runtime, as demonstrated across the Kafka and MongoDB repositories and the JSON, CSV, and Parquet formats evaluated here. We demonstrated that metadata-driven connector selection was practically viable at industrial scale. The Asset Cataloging registry stored connector identifiers, connection parameters, and routing rules as executable specifications, enabling the platform to invoke the correct connector at runtime without hardcoded integration logic or infrastructure replacement. The standardized four-operation connector interface maintained flexibility across heterogeneous repositories without prohibitive overhead; Parquet format flows achieved sub-100 ms average latency (avg 71 ms, P99 157 ms), confirming that the abstraction introduced negligible overhead independent of format choice. Our approach scaled to industrial requirements, processing 137.2 million real mining records at 10,000 Hz sensor sampling with no observed message loss under at-least-once delivery semantics. Four pilot sites subsequently deployed the platform in production, providing ongoing real-world validation.

Our design principles extend beyond mining to any domain where integration complexity exceeds processing complexity and infrastructure preservation is mandated. Separating orchestration (metadata) from execution (connectors) achieves extensibility while keeping existing repositories intact.

8.1. Limitations

The evaluation ran on a single virtual machine where all services shared one host, eliminating inter-host network latency; reported figures are therefore a lower bound for distributed deployments. For a representative LAN deployment, adding 1–10 ms inter-host round-trip yields an estimated 6–50 ms range, still within industrial IoT tolerances. The mining domain provides a rich but narrow validation context; domains with rapidly evolving schemas or higher message rates may expose limitations not visible here. Formal evaluation under multi-node distributed deployment, cross-site network delay, and large-scale concurrent streams remains future work. Security mechanisms (RBAC, TLS, audit logging) were exercised at pilot sites but not systematically benchmarked; a controlled security evaluation—penetration testing, adversarial workload benchmarking, and quantitative RBAC and audit overhead measurement—was outside this submission’s scope and is planned as future work, alongside fine-grained attribute-based access control and SIEM integration. Native exactly once delivery, via transactional connector operations for the subset of use cases that require it, is likewise left to future work; the current design attains equivalent end-state consistency for append-only workloads through idempotent writes. Finally, validation was limited to four pilot deployments in the mining and marine industries; evaluation on publicly available IIoT benchmark datasets and in additional sectors such as manufacturing, energy, and smart cities remains future work.

8.2. Future Work

The most important open evaluation gap is a direct, controlled comparison against general-purpose integration middleware (Apache NiFi, Kafka Connect, Apache Flink) under matched workloads. As discussed in Section 7, such a benchmark must isolate the storage-write path—identical sink, payload, and hardware—so that measured differences reflect orchestration overhead rather than the effort of hand-authoring flows; designing and running it, together with head-to-head throughput, latency, and operational-footprint measurements, is the primary target of our next evaluation phase.

Beyond benchmarking, we aim to extend executable metadata from a hand-authored artifact toward a largely self-describing integration layer. Automated connector generation from API specifications (OpenAPI, GraphQL schemas) and classification-based approaches [31,55] for automatic annotation of unlabeled sources could reduce development burden. Recent advances in large language models further open the prospect of LLM-assisted metadata authoring, drafting connector specifications and asset descriptions directly from documentation or sample payloads. Furthermore, integrating schema registries as metadata-aware components could enable automatic reconciliation, while ontology-based semantic extensions could support cross-domain data fusion. Industrial deployments also need fine-grained access control beyond basic authentication, motivating policy-aware orchestration in which access policies are themselves executable metadata applied at runtime.

Author Contributions: Conceptualization, M.K., G.K. and J.S.; methodology, M.K.; software, D.S.; validation, D.S. and G.K.; formal analysis, G.K. and M.K.; resources and investigation, M.F. and H.M.; writing—original draft preparation, D.S.; writing—review and editing, M.K., M.F., H.M., G.K. and J.S.; supervision, M.K., G.K. and J.S.; project administration, M.K., G.K. and J.S.; funding acquisition, G.K. and J.S. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the European Union’s Horizon Europe research and innovation programme under grant agreement No. 101091885 (MINE.IO—A Holistic Digital Mine 4.0 Ecosystem). The work presented in this paper was conducted as part of Work Package 5, Task 5.4 (IIoT Connection and Edge to Cloud Connection Platform), which developed the data integration infrastructure layer for the MINE.IO platform. Project website: <https://mineio-horizon.eu> (accessed on 15 May 2026).

Data Availability Statement: The data and source code presented in this study are available on reasonable request from the corresponding author.

Acknowledgments: The authors acknowledge the support of the MINE.IO project consortium (Grant Agreement No. 101091885) and the collaborative contributions of partners organizations across Europe. We thank the operational pilot sites for providing real-world requirements, operational datasets, and validation environments that were essential to the platform’s development and validation at industrial scale.

Conflicts of Interest: Author John Soldatos and Georgios Kouzas were contracted by Innov-Acts Limited. The remaining authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript; or in the decision to publish the results.

Abbreviations

The following abbreviations are used in this manuscript:

IoT	Internet of Things
IIoT	Industrial Internet of Things
API	Application Programming Interface
REST	Representational State Transfer
SOA	Service-Oriented Architecture
CAP	Consistency, Availability, Partition Tolerance
CRUD	Create, Read, Update, Delete
JSON	JavaScript Object Notation
CSV	Comma-Separated Values
HTTP	Hypertext Transfer Protocol
ETL	Extract, Transform, Load
UI	User Interface
WoT	Web of Things
RBAC	Role-Based Access Control
TLS	Transport Layer Security
DCAT	Data Catalog Vocabulary
UDP	User Datagram Protocol
NTP	Network Time Protocol
PTP	Precision Time Protocol
HDFS	Hadoop Distributed File System
WebHDFS	Web HDFS REST Interface
BDP	Big Data Platform
ERP	Enterprise Resource Planning
MES	Manufacturing Execution System
CRM	Customer Relationship Management
PLM	Product Lifecycle Management

References

1. Goel, S.S.; Goel, A.; Kumar, M.; Moltó, G. A review of Internet of Things: Qualifying technologies and boundless horizon. *J. Reliab. Intell. Environ.* **2021**, *7*, 23–33. <https://doi.org/10.1007/s40860-020-00127-w>.
2. Kumar, S.; Pragya, S.; Fatima, S.; Nigam, R. IoT Assisted Optimization of Resources for Healthcare, Mining and Transportation Industry. *Asian J. Res. Comput. Sci.* **2025**, *18*, 505–514. <https://doi.org/10.9734/ajrcos/2025/v18i4634>.
3. Qi, C.C. Big data management in the mining industry. *Int. J. Miner. Metall. Mater.* **2020**, *27*, 131–139. <https://doi.org/10.1007/s12613-019-1937-z>.
4. Li, J.; Liu, Y.; Xie, J.; Li, M.; Sun, M.; Liu, Z.; Jiang, S. A Remote Monitoring and Diagnosis Method Based on Four-Layer IoT Frame Perception. *IEEE Access* **2019**, *7*, 144324–144338. <https://doi.org/10.1109/access.2019.2945076>.

5. Aziz, A.; Schelén, O.; Bodin, U. A Study on Industrial IoT for the Mining Industry: Synthesized Architecture and Open Research Directions. *IoT* **2020**, *1*, 529–550. <https://doi.org/10.3390/iot1020029>.
6. Fan, J.; Han, F.; Liu, H. Challenges of Big Data Analysis. *Natl. Sci. Rev.* **2014**, *1*, 293–314. <https://doi.org/10.1093/nsr/nwt032>.
7. Wang, J.; Ma, Y.; Zhang, L.; Gao, R.X.; Wu, D. Industrial Big Data Analytics: Challenges, Methodologies, and Applications. *J. Manuf. Syst.* **2018**, *48*, 144–156. <https://doi.org/10.1016/j.jmsy.2018.01.003>.
8. Young, A.; Rogers, P. A review of digital transformation in mining. *Mining, Metall. Explor.* **2019**, *36*, 683–699. <https://doi.org/10.1007/s42461-019-00103-w>.
9. Tu, D.Q.; Kayes, A.S.M.; Rahayu, W.; Nguyen, K. IoT streaming data integration from multiple sources. *Computing* **2020**, *102*, 2299–2329. <https://doi.org/10.1007/s00607-020-00830-9>.
10. Reda, R.; Piccinini, F.; Martinelli, G.; Carbonaro, A. Heterogeneous self-tracked health and fitness data integration and sharing according to a linked open data approach. *Computing* **2021**, *104*, 835–857. <https://doi.org/10.1007/s00607-021-00988-w>.
11. Vavilapalli, V.K.; Murthy, A.C.; Douglas, C.; Agarwal, S.; Konar, M.; Evans, R.; Graves, T.; Lowe, J.; Shah, H.; Seth, S.; et al. Apache Hadoop YARN: Yet Another Resource Negotiator. In Proceedings of the 4th Symposium on Cloud Computing (SOCC), Santa Clara, CA, USA, 1–3 October 2013; pp. 1–16. <https://doi.org/10.1145/2523616.2523633>.
12. Zaharia, M.; Xin, R.S.; Wendell, P.; Das, T.; Armbrust, M.; Dave, A.; Meng, X.; Rosen, J.; Venkataraman, S.; Franklin, M.J.; et al. Apache Spark: A Unified Engine for Big Data Processing. *Commun. ACM* **2016**, *59*, 56–65. <https://doi.org/10.1145/2934664>.
13. Apache Software Foundation. Apache NiFi System Administrator's Guide. Dataflow Automation Platform for Routing, Transformation, and System Mediation Logic. 2024. Available online: <https://nifi.apache.org/docs.html> (accessed on 15 May 2026).
14. Apache Software Foundation. Apache Kafka Connect Documentation. Framework for Scalable and Reliable Streaming Data Movement Between Kafka and Other Systems. 2024. Available online: <https://kafka.apache.org/documentation/#connect> (accessed on 15 May 2026).
15. McCool, M.; Kovatsch, M.; Toumura, K.; Nimura, K.; Kabisch, S.; Reshetova, E. *Web of Things (WoT) Architecture 1.1; W3c Recommendation; W3C Architectural Framework for IoT Interoperability Using Metadata-Driven Thing Descriptions; World Wide Web Consortium (W3C)*; Wakefield, MA, USA, 2023.
16. Cirillo, F.; Solmaz, G.; Berz, E.; Bauer, M.; Cheng, B.; Kovacs, E. A Standard-Based Open Source IoT Platform: FIWARE. *IEEE Internet Things Mag.* **2019**, *2*, 12–18. <https://doi.org/10.1109/IOTM.0001.1800022>.
17. Costa, F.; Nassar, S.; Gusmeroli, S.; Schultz, R.; Conceição, A.S.; Xavier, M.G.; Hessel, F.; Dantas, M. FASTEN IIoT: An Open Real-Time Platform for Vertical, Horizontal and End-To-End Integration. *Sensors* **2020**, *20*, 5499. <https://doi.org/10.3390/s20195499>.
18. Abu-Elkheir, M.; Hayajneh, M.; Ali, N. Data Management for the Internet of Things: Design Primitives and Solution. *Sensors* **2013**, *13*, 15582–15612. <https://doi.org/10.3390/s131115582>.
19. Chen, M.; Mao, S.; Liu, Y. Big data: A survey. *Inf. Sci.* **2014**, *275*, 314–347. <https://doi.org/10.1016/j.ins.2014.01.015>.
20. Zaharia, M.; Chowdhury, M.; Franklin, M.J.; Shenker, S.; Stoica, I. Spark: Cluster computing with working sets. In Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing, HotCloud'10, Boston, MA, USA, 22–25 June 2010; p. 10.
21. Carbone, P.; Katsifodimos, A.; Ewen, S.; Markl, V.; Haridi, S.; Tzoumas, K. Apache flink: Stream and batch processing in a single engine. *Bull. Tech. Comm. Data Eng.* **2015**, *38*.
22. Alexandrov, A.; Bergmann, R.; Ewen, S.; Freytag, J.C.; Hueske, F.; Heise, A.; Kao, O.; Leich, M.; Leser, U.; Markl, V.; et al. The Stratosphere Platform for Big Data Analytics. *VLDB J.* **2014**, *23*, 939–964. <https://doi.org/10.1007/s00778-014-0357-y>.
23. Putrama, I.M.; Martinek, P. Heterogeneous data integration: Challenges and opportunities. *Data Brief* **2024**, *56*, 110853. <https://doi.org/10.1016/j.dib.2024.110853>.
24. Ramadhan, H.; Indikawati, F.I.; Kwon, J.; Koo, B. MusQ: A Multi-Store Query System for IoT Data Using a Datalog-Like Language. *IEEE Access* **2020**, *8*, 58032–58056. <https://doi.org/10.1109/access.2020.2982472>.
25. Duggan, J.; Elmore, A.J.; Stonebraker, M.; Balazinska, M.; Cetintemel, U.; Gadepally, V.; Zdonik, S. The BigDAWG polystore system. *ACM Sigmod Rec.* **2015**, *44*, 11–16. <https://doi.org/10.1145/2814710.2814713>.
26. Kirmse, A.; Kraus, V.; Hoffmann, M.; Meisen, T. An Architecture for Efficient Integration and Harmonization of Heterogeneous, Distributed Data Sources Enabling Big Data Analytics. In Proceedings of the International Conference on Enterprise Information Systems (ICEIS), Funchal, Portugal, 21–24 March 2018; pp. 175–182. <https://doi.org/10.5220/0006776701750182>.
27. Wang, F.; Hu, L.; Zhou, J.; Zhao, K. A Data Processing Middleware Based on SOA for the Internet of Things. *J. Sens.* **2015**, *2015*, 827045:1–827045:8. <https://doi.org/10.1155/2015/827045>.
28. Sejdiu, B.; Ismaili, F.; Ahmedi, L. IoT SAS: An Integrated System for Real-Time Semantic Annotation and Interpretation of IoT Sensor Stream Data. *Computers* **2021**, *10*, 127. <https://doi.org/10.3390/computers10100127>.
29. Huang, X.; Fan, J.; Deng, Z.; Yan, J.; Li, J.; Wang, L. Efficient IoT Data Management for Geological Disasters Based on Big Data-Turbocharged Data Lake Architecture. *ISPRS Int. J. Geo-Inf.* **2021**, *10*, 743. <https://doi.org/10.3390/ijgi10110743>.
30. Azzabi, W.; Zemmar, A.; Drias, H. Data Lakes: A Survey of Concepts and Architectures. *Computers* **2024**, *13*, 183. <https://doi.org/10.3390/computers13070183>.

31. Montori, F.; Liao, K.; Giosa, M.D.; Jayaraman, P.; Bononi, L.; Sellis, T.; Georgakopoulos, D. A Metadata-Assisted Cascading Ensemble Classification Framework for Automatic Annotation of Open IoT Data. *IEEE Internet Things J.* **2023**, *10*, 13401–13413. <https://doi.org/10.1109/jiot.2023.3263213>.
32. Gouripeddi, R.; Tran, L.T.T.; Madsen, R.; Gangadhar, T.; Mo, P.; Burnett, N.; Butcher, R.; Sward, K.; Facelli, J. An Architecture for Metadata-driven Integration of Heterogeneous Sensor and Health Data for Translational Exposomic Research. In Proceedings of the 2019 IEEE EMBS International Conference on Biomedical & Health Informatics (BHI), Chicago, IL, USA, 19–22 May 2019; pp. 1–4. <https://doi.org/10.1109/bhi.2019.8834657>.
33. Guo, X.; Ma, L.; Su, K.; Sun, Y.F. Research and Design of Multi-Source Heterogeneous Information Integration Platform for Metadata Service. *Appl. Mech. Mater.* **2014**, *513–517*, 1485–1489. <https://doi.org/10.4028/www.scientific.net/amm.513-517.1485>.
34. Hong, J.H.; Shi, Y.T. Integration of Heterogeneous Sensor Systems for Disaster Responses in Smart Cities: Flooding as an Example. *ISPRS Int. J. Geo-Inf.* **2023**, *12*, 279. <https://doi.org/10.3390/ijgi12070279>.
35. Bailo, D.; Paciello, R.; Vinciarelli, V.; Orfino, A.; Jeffery, K.G.; Michalek, J.; Goffi, C. Integrated Access to Multidisciplinary Data Through Semantically Interoperable Services in a Metadata-Driven Platform for Solid Earth Science. In *Communications in Computer and Information Science*; Springer Nature: Geneva, Switzerland, 2023; pp. 235–247. https://doi.org/10.1007/978-3-031-39141-5_20.
36. Le-Phuoc, D.; Nguyen-Mau, H.Q.; Parreira, J.; Hauswirth, M. A Middleware Framework for Scalable Management of Linked Streams. *J. Web Semant.* **2012**, *16*, 42–51. <https://doi.org/10.1016/j.websem.2012.06.003>.
37. Rucco, C.; Longo, A.; Saad, M. MIND: A metadata-driven INgestion design pattern for efficient data ingestion. *Big Data Res.* **2026**, *43*, 100574. <https://doi.org/10.1016/j.bdr.2025.100574>.
38. Sejdiu, B.; Ismaili, F.; Ahmedi, L. A Management Model of Real-time Integrated Semantic Annotations to the Sensor Stream Data for the IoT. In Proceedings of the 16th International Conference on Web Information Systems and Technologies (WEBIST 2020), Virtually, 3–5 November 2020; pp. 59–66. <https://doi.org/10.5220/0010111500590066>.
39. Muñoz, M.; Torres, M.; Gil, J.; Guzmán, J. An Internet of Things platform for heterogeneous data integration: Methodology and application examples. *J. Netw. Comput. Appl.* **2025**, *240*, 104197. <https://doi.org/10.1016/j.jnca.2025.104197>.
40. Bicaku, A.; Maksuti, S.; Hegedus, C.; Tauber, M.; Delsing, J.; Eliasson, J. Interacting with the Arrowhead Local Cloud: Onboarding Procedure. In Proceedings of the 2018 IEEE International Conference on Industrial Cyber-Physical Systems (ICPS), Saint Petersburg, Russia, 15–18 May 2018; pp. 743–748. <https://doi.org/10.1109/ICPHYS.2018.8390800>.
41. Martin, R.A.; Lin, S.-W.; Miller, B.; Durand, J.; Joshi, R.; Didier, P.; Chigani, A.; Torenbeek, R.; Duggal, D.; Bleakley, G.; et al. Industrial Internet Reference Architecture Technical Report. 2015. Available online: https://www.researchgate.net/publication/311948151_Industrial_Internet_Reference_Architecture_Technical_Report?channel=doi&linkId=5863e5b608ae8fce490b6f17&showFulltext=true (accessed on 15 May 2026).
42. Martin, D.; Kuhl, N.; Schwenk, M. Towards a Reference Architecture for Future Industrial Internet of Things Networks. In *Proceedings of the 2021 IEEE 23rd Conference on Business Informatics (CBI)*; IEEE: Piscataway, NJ, USA, 2021; pp. 1–9. <https://doi.org/10.1109/cbi52690.2021.10049>.
43. Valtolina, S.; Ferrari, L.; Mesiti, M. Ontology-Based Consistent Specification of Sensor Data Acquisition Plans in Cross-Domain IoT Platforms. *IEEE Access* **2019**, *7*, 176141–176169. <https://doi.org/10.1109/access.2019.2957855>.
44. Kiljander, J.; D'Elia, A.; Morandi, F.; Hyttinen, P.; Takalo-Mattila, J.; Ylisaukko-oja, A.; Soininen, J.; Cinotti, T.S. Semantic Interoperability Architecture for Pervasive Computing and Internet of Things. *IEEE Access* **2014**, *2*, 856–873. <https://doi.org/10.1109/ACCESS.2014.2361036>.
45. Cimmino, A.; Poveda-Villalón, M.; García-Castro, R. eWoT: A Semantic Interoperability Approach for Heterogeneous IoT Ecosystems Based on the Web of Things. *Sensors* **2020**, *20*, 822. <https://doi.org/10.3390/s20030822>.
46. Jin, W.; Xu, R.; Lim, S.; Park, D.H.; Park, C.; Kim, D. Integrated Service Composition Approach Based on Transparent Access to Heterogeneous IoT Networks Using Multiple Service Providers. *Mob. Inf. Syst.* **2021**, *2021*, 5590605. <https://doi.org/10.1155/2021/5590605>.
47. Niknejad, N.; Ismail, W.; Ghani, I.; Nazari, B.; Bahari, M.; Hussin, A.R.B.C. Understanding service-oriented architecture (SOA): A systematic literature review and directions for further investigation. *Inf. Syst.* **2020**, *91*, 101491. <https://doi.org/10.1016/j.is.2020.101491>.
48. Kruchten, P. The 4 + 1 View Model of architecture. *IEEE Softw.* **1995**, *12*, 42–50. <https://doi.org/10.1109/52.469759>.
49. Rodríguez, C.; Baez, M.; Daniel, F.; Casati, F.; Trabucco, J.C.; Canali, L.; Percannella, G. REST APIs: A Large-Scale Analysis of Compliance with Principles and Best Practices. In *Proceedings of the Web Engineering—16th International Conference, ICWE 2016, Lugano, Switzerland, 6–9 June 2016*; Proceedings; Bozzon, A., Cudré-Maroux, P., Pautasso, C., Eds.; Lecture urls in Computer Science; Springer International Publishing: Berlin/Heidelberg, Germany, 2016; Volume 9671, pp. 21–39. https://doi.org/10.1007/978-3-319-38791-8_2.
50. Chamari, L.; Petrova, E.; Pauwels, P. An End-to-End Implementation of a Service-Oriented Architecture for Data-Driven Smart Buildings. *IEEE Access* **2023**, *11*, 117261–117281. <https://doi.org/10.1109/access.2023.3325767>.
51. Hankel, M.; Rexroth, B. The reference architectural model industrie 4.0 (rami 4.0). *Zvei* **2015**, *2*, 4–9.

52. Lymperis, D.; Goumopoulos, C. SEDIA: A Platform for Semantically Enriched IoT Data Integration and Development of Smart City Applications. *Future Internet* **2023**, *15*, 276. <https://doi.org/10.3390/fi15080276>.
53. Peng, C.; Goswami, P. Meaningful Integration of Data from Heterogeneous Health Services and Home Environment Based on Ontology. *Sensors* **2019**, *19*, 1747. <https://doi.org/10.3390/s19081747>.
54. Jachimczyk, B.; Tkaczyk, R.; Piotrowski, T.; Johansson, S.; Kulesza, W. IoT-based Dairy Supply Chain—An Ontological Approach. *Elektron. Elektrotechnika* **2021**, *27*, 71–83. <https://doi.org/10.5755/j02.eie.27612>.
55. Montori, F.; Liao, K.; Jayaraman, P.; Bononi, L.; Sellis, T.; Georgakopoulos, D., Classification and Annotation of Open Internet of Things Datastreams. In *Lecture Notes in Computer Science*; Springer International Publishing: Berlin/Heidelberg, Germany, 2018; pp. 209–224. https://doi.org/10.1007/978-3-030-02925-8_15.

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.