International Journal of Computational and Experimental Science and ENgineering (IJCESEN)

Vol. 12-No.1 (2026) pp. 964-976
http://www.ijcesen.com

ISSN: 2149-9144

Research Article

A Dynamic Management System of Synchronous-Asynchronous API Services

Hichem Chaalal[1]* Fouad Khatemi[2], Mohamed Amine Chemrak[3]

[1]University of Tissemsilt, Algeria
* **Corresponding Author Email:** hichem.chaalal@univ-tissemsilt.dz- **ORCID:** 0000-0002-5887-7850

[2]University of Tissemsilt, Algeria
Email: foua2d@gmail.com- **ORCID:** 0000-0002-5247-0850

[3]University of Tissemsilt, Algeria
Email: amin2a@gmail.com - **ORCID:** 0000-0002-5247-0050

Article Info:

DOI: 10.22399/ijcesen.4953
Received : 14 October 2025
Revised : 20 January 2026
Accepted : 22 February 2026

Keywords

API aggregation,
synchronous-asynchronous hybrid,
Redis,
dynamic data ingestion,
schema-less querying,
Kafka

Abstract:

Modern applications increasingly rely on data from heterogeneous APIs, databases, and IoT sources, yet existing integration solutions struggle with latency, schema heterogeneity, data consistency, and fault tolerance. This paper proposes a hybrid synchronous-asynchronous API management system that enables low-latency interactive access and high-throughput background processing. The core innovation is automatic, schema-agnostic ingestion of JSON responses into endpoint-specific Redis tables, augmented with reception timestamps, enabling efficient time-series and historical querying without manual ETL or schema inference. The system exposes a REST/WebSocket interface for synchronous requests, a Kafka-based asynchronous pipeline for complex aggregations, and a SQL-like query layer over the stored data. Evaluation on a Kubernetes cluster with simulated heterogeneous APIs demonstrates median response times of 78 ms for synchronous aggregations (35–62% faster than GraphQL federation and REST aggregation baselines) and 99.2% data completeness under 30% concurrent source failure in asynchronous mode. Case studies in smart-city dashboards and healthcare monitoring illustrate reduced development effort and improved performance. The architecture bridges API consumption and database functionality, offering a lightweight alternative for real-time federated analysis and IoT integration.

1. Introduction

Contemporary software systems depend heavily on data from diverse, distributed sources including REST/GraphQL APIs, streaming platforms, and IoT devices. This heterogeneity introduces significant challenges: variable latency, inconsistent schemas, over/under-fetching, partial failures, and complex ETL pipelines required to enable unified querying.

Traditional approaches fall short in several respects. Sequential REST calls incur high latency and network overhead. GraphQL federation reduces over-fetching but requires predefined schemas and resolver logic, making it brittle under schema evolution. Event-driven platforms such as Kafka excel at high-throughput processing but lack native support for synchronous low-latency access or queryable historical storage.

This work introduces a hybrid API management system that combines:

- Synchronous mode for interactive, low-latency aggregations with adaptive timeouts and partial results
- Asynchronous mode for high-volume, long-running tasks via Kafka
- Automatic ingestion of JSON responses into endpoint-organized, timestamped Redis tables
- Schema-agnostic storage that defers structural interpretation to query time
- A SQL-like interface for ad-hoc querying across aggregated data

By eliminating manual schema definition and ETL while supporting both interaction patterns in a single architecture, the system reduces integration complexity and enables efficient real-time and historical analysis. The remainder of the paper

reviews related work, details the system design and implementation, presents evaluation results, and discusses limitations and future directions.

2. Related Work

Modern data integration systems face significant challenges when aggregating heterogeneous data from distributed sources. This section reviews existing approaches across four key areas: API aggregation, schema inference and normalization, dynamic query capabilities, and microservices architectures with API gateways.

2.1 API Aggregation Approaches

Traditional REST aggregation relies on sequential client requests with manual result joining, which introduces significant latency and network overhead as data sources increase [7]. Backend-for-Frontend (BFF) patterns provide client-specific aggregation endpoints but still require manually implemented aggregation logic for each endpoint [8]. GraphQL [6] addresses over-fetching by allowing clients to specify required fields in a single request, while GraphQL federation [3] extends this capability across distributed services. However, GraphQL implementations face challenges with schema evolution and require substantial effort for schema definitions and resolver implementations [27]. Vázquez-Ingelmo et al. [26] demonstrate GraphQL's effectiveness in interoperability scenarios while noting challenges with caching and performance optimization.

Event-driven architectures [16] focus on asynchronous data flow. Platforms like Kafka [14] support high-throughput stream processing but typically lack built-in support for schema harmonization and synchronous interactions. Stream-based data integration methods can handle large volumes but present implementation and operational challenges [5]. The Lambda Architecture [15] attempts to combine batch and stream processing for comprehensive data integration but has been criticized for complexity and maintenance difficulties [13].

Recent industry and academic works emphasize hybrid synchronous-asynchronous designs for scalable APIs. Modern gateways unify REST/GraphQL (synchronous) with Kafka/MQTT/WebSockets (asynchronous) [29, 30, 31]. Studies on event-driven microservices highlight Kafka's role in asynchronous patterns while noting trade-offs in real-time latency [34, 39].

2.2 Schema Inference and Normalization

Schema inference for heterogeneous data sources has been extensively studied in academia and industry. Rahm and Bernstein [22] provide a comprehensive overview of automatic schema matching methods, categorizing them by their use of schema information, instance data, and external resources. More recently, Baazizi et al. [1] developed methods for inferring schemas from large JSON datasets, addressing challenges posed by schema evolution and structural variations.

Gallinucci et al. [9] discuss schema profiling for document-oriented databases, extending beyond basic type inference to include distribution statistics and semantic correlations. Their approach demonstrates improved accuracy for schema interpretation but requires significant computational resources for large datasets. In the industrial domain, MongoDB's schema suggestion features [2] provide basic inferences for document collections but lack deep semantic understanding.

Recent advances in machine learning have enabled more sophisticated approaches to schema inference and data integration. Mudgal et al. [17] investigated deep learning methods for entity matching, demonstrating superior performance over traditional methods in complex scenarios. Cappuzzo et al. [4] present methods for creating embeddings of heterogeneous relational databases, enabling semantic matching across disparate schemas. While promising for automated schema harmonization, these approaches have not yet seen widespread adoption in production API aggregation systems.

The Data Civilizer system [11] provides a comprehensive approach to data discovery and integration using both automated schema inference and human feedback loops. This approach is powerful but requires substantial infrastructure and expertise. Unlike these approaches, our system bypasses schema inference entirely by storing raw JSON messages in Redis with endpoint-based table organization and timestamps. This simplifies data integration and eliminates the computational overhead of inference processes.

2.3 Dynamic Query Capabilities

Dynamic querying of heterogeneous data sources presents unique challenges. JSONata [25] is a query and transformation language specifically designed for JSON data, operating similarly to XPath and XSLT but tailored for JSON structures. While excellent for single-document transformations, it lacks support for cross-document joins and aggregations needed for comprehensive data integration.

In the database domain, Hybrid Transactional/Analytical Processing (HTAP) systems [20] attempt to support both operational and analytical workloads within a single architecture. Systems like TiDB [10] and MemSQL [24] enable SQL queries on distributed storage layers, allowing complex queries with minimal latency. However, these systems typically require data loading with predefined schemas and ETL processes, limiting their utility for dynamic API aggregation.

Self-driving database management systems [21] represent a novel approach combining automated schema design, indexing, and query optimization. These systems can adapt to varying workloads and data characteristics, potentially addressing some challenges in heterogeneous data aggregation. However, current implementations primarily focus on traditional database workloads rather than API integration scenarios.

Redis-based architectures support low-latency JSON storage and querying in real-time systems [32, 35], often combined with Kafka for high-throughput ingestion [33, 37]. These complement schema-flexible approaches for heterogeneous data integration [37, 38]. Our system leverages Redis to store JSON messages in a queryable format, enabling dynamic querying without predefined schemas or complex ETL.

2.4 Microservices and API Gateways

Microservices architectures [18] have increased the importance of effective API aggregation solutions. As monolithic applications decompose into independently deployable services [19], the number of APIs clients must interact with grows substantially. This decomposition facilitates development and scaling but complicates data aggregation and consistency management.

API gateways have become a standard approach to manage communication in microservices environments [23]. These gateways serve as centralized entry points for client requests, handling cross-cutting concerns like authentication, rate limiting, and request routing. Some advanced API gateways offer limited aggregation capabilities, but these typically require manual configuration and lack sophisticated schema harmonization or dynamic table formation.

Indrasiri and Siriwardena [12] discuss data integration in microservice architectures using API composition, CQRS [28], and event sourcing. These patterns provide architectural guidance but still require substantial implementation effort and domain knowledge. Our work extends existing approaches by adding automated capabilities that reduce manual effort in microservices data aggregation, utilizing Redis for structured storage and querying.

Unlike inference-heavy methods [1, 9, 22] or protocol-focused gateways [30, 31], our system provides automatic, timestamped Redis table generation for API responses, enabling schema-agnostic querying without ETL—building on recent hybrid streaming pipelines [33, 39] and heterogeneous data strategies [37].

3. System Design

3.1 Architecture Overview

The proposed hybrid synchronous-asynchronous API architecture is organized into layered components that balance low-latency data access with scalable background processing and support for complex queries over aggregated data. The architecture comprises five primary layers: the Client Layer, the Aggregation Layer, the Table Generation Layer, the Async Processing Layer, and the Storage Layer. The Client Layer provides a unified interface through which applications interact with the system, regardless of whether the interaction is synchronous or asynchronous.

This layer exposes a REST API for conventional request-response interactions, a WebSocket interface for real-time updates and streaming results, and a SQL-like query endpoint for complex queries over aggregated data. The Aggregation Layer forms the orchestration core of the system. It retrieves data from multiple heterogeneous sources, normalizes it, and merges the results into a unified response. This layer supports concurrent request handling and incorporates robust timeout management to prevent slow sources from degrading overall system responsiveness.

3.2 *Synchronous Mode*

The synchronous mode is designed to deliver low-latency responses while maximizing data completeness and accuracy. When a client initiates a synchronous aggregation request, the system executes a structured orchestration procedure to optimize both response time and result quality. The request flow begins when a client issues a GET request to the "/sync" endpoint, specifying the desired data sources along with optional parameters such as timeout limits, priority levels, and data transformation rules. Upon receiving the request, the Aggregation Layer parses the parameters and constructs an execution plan that optimizes the aggregation process based on source characteristics and client requirements.

3.3 *Asynchronous Mode*

The asynchronous mode enables scalable data aggregation by decoupling request submission from result retrieval, allowing the system to handle complex, long-running, or high-volume aggregation tasks without blocking the client. This mode is particularly suited for computationally intensive transformations, large result sets, or sources with limited throughput.

The asynchronous workflow is initiated when a client submits a POST request to the "/async" endpoint, providing a task specification that describes the target data sources, transformation rules, and delivery preferences. Unlike synchronous mode, which requires all parameters to be supplied upfront in a single request, asynchronous mode supports richer task definitions that may incorporate conditional logic, multi-stage processing pipelines, and sophisticated error handling procedures.

3.4 Dynamic Table Generation

Dynamic table generation is a core innovation of the proposed system, bridging the gap between API consumption and database functionality. The system automatically constructs structured, queryable tables from heterogeneous API responses by persisting raw JSON messages in Redis. Each message originating from the same endpoint is stored in a dedicated table, augmented with a timestamp indicating the time of reception. This approach eliminates the need for manual ETL operations or AI-assisted schema inference, allowing users to issue complex queries over the aggregated data through a SQL-like interface that operates directly on the stored JSON structure.

The process begins when an API response is received. The system extracts the JSON payload, identifies the source endpoint, and stores the message in a Redis table dedicated to that endpoint. A timestamp is automatically appended to each entry. This approach ensures that data is organized by source and time, enabling efficient time-based querying and historical analysis. The system does not perform schema inference; instead, it allows queries to be executed directly on the JSON structure, providing flexibility and reducing computational overhead.

4. Implementation

4.1 Technology Stack

The technology stack for the hybrid synchronous-asynchronous API system was selected to achieve an effective balance between performance, reliability, and developer productivity. Each component was chosen based on its specific capabilities and its complementary role within the overall architecture, with the goal of addressing the inherent challenges of heterogeneous data integration.

FastAPI is a modern Python web framework that supports asynchronous request handling via Python's async/await syntax and is used to implement the API Gateway layer. FastAPI was selected over alternatives such as Flask or Django due to its superior performance in I/O-bound scenarios, which are characteristic of API aggregation workloads. Apache Kafka serves as the primary message broker for the streaming and asynchronous processing components. For development environments and smaller production deployments, Redpanda is used as a lightweight alternative. Kafka's distributed architecture ensures durability and scalability for asynchronous processing, and its partitioning strategy enables concurrent execution of multiple aggregation workers.

Redis is employed as the primary storage layer for dynamic table generation. It stores JSON messages organized by endpoint, with each entry including a timestamp. Redis's in-memory nature provides low-latency access, while its support for JSON data structures allows efficient querying. For persistent storage and complex querying, PostgreSQL is used alongside Redis to store job metadata, results, and generated table schemas.

4.2 Key Algorithms

The performance characteristics of the hybrid API system are underpinned by a set of purpose-designed algorithms that address the core challenges of heterogeneous data aggregation. These algorithms represent a key technical contribution of this work, combining established techniques with novel strategies to improve both performance and adaptability. The JSON Message Storage and Retrieval algorithm governs the persistence of incoming API responses in Redis, organizing entries by source endpoint and reception timestamp. This eliminates the need for schema inference at ingestion time and enables efficient time-based querying over stored records.

4.3 Algorithm Examples

The following algorithm examples illustrate key components of our implementation, demonstrating how the architectural principles and algorithms are realized in practice. These examples have been

simplified for clarity while preserving the essential logic and patterns.

The synchronous aggregation handler implements the core request-response flow for immediate data access.

Algorithm 1: JSON Message Storage in Redis

Input:

json_message: JSON object from API response.
endpoint: Source endpoint identifier.
timestamp: Time of receipt.

Output:

Stored message in Redis with endpoint-based key and timestamp.

Algorithm Store_Message(json_message, endpoint, timestamp)

1. // **Generate Redis key based on endpoint**
2. redis_key ← "api:" + endpoint + ":data"
3. // **Append timestamp to JSON message**
4. json_message["_timestamp"] ← timestamp
5. // **Store message in Redis list or sorted set**
6. Redis.RPush(redis_key, json_message)
7. // **Optional: Store in sorted set for time-range queries**
8. Redis.ZAdd(redis_key + ":sorted", timestamp, json_message)
9. **Return** redis_key

End Algorithm

Algorithm 2: Synchronous Aggregation with Caching and Adaptive Timeout

Input:

sources: List of data sources.
timeout: Maximum allowed response time (default = 5.0 seconds).
priority: Priority values for sources (optional).
cache: Boolean flag indicating whether caching is enabled (default = True).

Output:

Aggregated, normalized, and optionally cached dataset.

Algorithm Sync_Aggregate(sources, timeout = 5.0, priority = None, cache = True)

1. // **Step 1: Normalize and validate sources**
2. validated_sources ← Validate_Sources(sources)
3. // **Step 2: Apply default priorities if not provided**
4. **If** priority = None OR length(priority) ≠ length(sources) **then**
5. priority ← [1, 1, ... , 1] // uniform priority assignment
6. **EndIf**
7. // **Step 3: Check cache (if enabled)**
8. **If** cache = True **then**
9. cache_key ← Generate_Cache_Key(sources, priority)
10. cached_result ← Redis_Get(cache_key)
11. **If** cached_result ≠ ∅ **then**
12. Return Deserialize(cached_result)
13. **EndIf**
14. **EndIf**
15. // **Step 4: Concurrent data fetching with adaptive timeouts**
16. history ← Get_Performance_History(sources)
17. results ← Fetch_With_Adaptive_Timeout(validated_sources, priority, history)
18. // **Step 5: Store messages in Redis with timestamps**
19. **For each** result ∈ results **do**
20. Store_Message(result.json, result.endpoint, Current_Time())
21. **EndFor**
22. // **Step 6: Normalize and merge results**
23. normalized ← Normalize_Results(results)
24. merged ← Merge_Normalized_Data(normalized)
25. // **Step 7: Cache results (if enabled)**
26. **If** cache = True **then**
27. Redis_Set(cache_key, Serialize(merged), Expiry = Calculate_Cache_TTL(sources))
28. **EndIf**
29. Return merged

End Algorithm

Algorithm 3: Asynchronous Job Processing with Data Aggregation and Error Handling

Input:

kafka_client: Client interface for receiving aggregation job messages.
storage_manager: Component responsible for storing job status and results.
job_spec: Specification of a job containing sources, transformations, and optional metadata (e.g., table creation, notifications).

Output:

Processed job results stored in persistent storage, along with updated job status and optional notifications.

Algorithm Async_Job_Processor(kafka_client, storage_manager)

1. // **Initialization**
2. running ← False
3.
4. **Procedure Start()**
5. running ← True
6. Subscribe(kafka_client, "aggregation_jobs")
7. **While** running = True **do**
8. messages ← kafka_client.Poll(timeout = 1000 ms)
9. **For each** message ∈ messages **do**
10. Process_Job(message.value)
11. **EndFor**

EndWhile
EndProcedure

Procedure Process_Job(job_spec)
job_id ← job_spec["id"]
Try
// **Step 1: Update job status**
storage_manager.Update_Status(job_id, "processing")

// **Step 2: Fetch data from sources**
results ← ∅
For each source ∈ job_spec["sources"] do
Try
data ← Fetch_Data(source["url"], headers = source.get("headers", ∅), params = source.get("params", ∅))
results[source["id"]] ← data
// **Step 2a: Store each message in Redis**
Store_Message(data, source["id"], Current_Time())
Catch Exception e
results[source["id"]] ← { "error" : e.message }
EndTry
EndFor

// **Step 3: Apply transformations if defined**
If "transformations" ∈ job_spec **then**
results ← Apply_Transformations(results, job_spec["transformations"])
EndIf

// **Step 4: Store results**
result_id ← storage_manager.Store_Results(job_id, results)

// **Step 5: Update job status to completed**
storage_manager.Update_Status(job_id, "completed", result_id = result_id)

// **Step 6: Trigger success notifications if configured**
If "notifications" ∈ job_spec **then**
Send_Notifications(job_spec["notifications "], job_id, "completed")
EndIf

Catch Exception e
// **Step 7: Handle job failure**
storage_manager.Update_Status(job_id, "failed", error = e.message)

// **Step 8: Trigger failure notifications if configured**
If "notifications" ∈ job_spec **then**
Send_Notifications(job_spec["notifications "], job_id, "failed", error = e.message)
EndIf
EndTry
EndProcedure

End Algorithm

4.4 Deployment Architecture

The proposed hybrid system can be deployed across a range of environments, from single-node development instances to large-scale production clusters. The reference deployment uses Kubernetes as the orchestration platform, enabling automatic scaling, self-healing, and declarative configuration. The deployment architecture cleanly separates stateless and stateful components, allowing each to scale independently in response to workload demands. The API Gateway and Aggregation Layer are deployed as stateless services that scale horizontally to accommodate increased request volumes. These components are configured with explicit resource requests and limits to ensure appropriate CPU and memory allocation and to prevent resource exhaustion under peak load. The Async Processing Layer leverages Kafka's partitioning model to distribute processing across multiple worker instances. Each worker subscribes to a dedicated set of partitions, enabling concurrent processing while preserving intra-partition ordering. This design allows the system to increase throughput by adding additional worker instances without modifying the job submission or result retrieval interfaces. For the Storage Layer, a tiered strategy is employed to balance performance and cost: Redis provides low-latency access for JSON message storage and real-time querying, while PostgreSQL offers durable persistence for job results and metadata. This combination ensures rapid access to recent data while guaranteeing long-term retention of historical results.

4.5 Querying Evolving Schemas in Document-Oriented Tables

A central challenge in aggregating data from heterogeneous API sources is **schema evolution**. API providers frequently introduce new attributes, deprecate existing fields, or modify nesting structures over time. In traditional relational database systems, such changes typically require explicit schema migrations and coordinated updates to extract–transform–load (ETL) pipelines, which increases operational complexity and may lead to service disruption. Even automated schema inference techniques struggle to adapt efficiently to

continuously evolving data formats, particularly in large-scale and real-time environments [31].

The proposed system adopts a document-oriented storage model that natively accommodates schema variability. Incoming API responses are stored as JSON documents in Redis without enforcing a predefined schema. Documents originating from the same endpoint are grouped into a logical table and augmented with a timestamp indicating the time of reception. As a result, multiple schema versions originating from the same API endpoint may coexist within the same table, accurately reflecting the natural evolution of external data sources.

Querying over such heterogeneous collections is enabled through a **schema-tolerant SQL-like query interface** that operates directly on the JSON document structure. Query evaluation is performed only on fields present within each document, while documents lacking the queried attributes are safely ignored. This behavior prevents query failures caused by missing or renamed fields and enables stable query execution across mixed schema versions. Structural variations introduced by API evolution can be handled at query time using conditional expressions, projections, and alternative field mappings, thereby preserving backward compatibility.

Unlike prior approaches that rely on explicit schema inference, profiling, or normalization techniques [9], [31], the proposed architecture deliberately **bypasses schema inference at ingestion time**. This design choice reduces computational overhead, minimizes ingestion latency, and avoids the brittleness associated with inferred global schemas that may rapidly become obsolete. Instead, structural interpretation is deferred to the query layer, where adaptability can be explicitly managed according to application-specific requirements.

This strategy enables efficient time-based querying, historical analysis across schema versions, and flexible aggregation of evolving API data without requiring data migration or reprocessing. Consequently, the system is well suited for real-time dashboards, IoT data integration, and federated analytics scenarios, where schema instability is intrinsic and continuous evolution is the norm.

5. Evaluation

5.1 Experimental Setup

A comprehensive evaluation was conducted under conditions designed to replicate realistic deployment scenarios while maintaining controlled and reproducible experimental conditions. This section describes the experimental methodology, including the hardware infrastructure, software configuration, simulated data sources, workload profiles, and performance metrics employed throughout the evaluation. Experiments were conducted on a Kubernetes cluster comprising eight worker nodes, each equipped with 16 Intel Xeon E5-2680 v4 CPU cores, 64 GB of RAM, and NVMe SSD storage. The network interconnect operated at 10 Gbps with an average round-trip latency of 0.5 ms. Ten simulated API endpoints were constructed to represent a diverse set of heterogeneous data sources with realistic behavioral profiles. These endpoints varied across a range of latency and reliability characteristics: some exhibited minimal latency and high availability, while others were configured with variable response times and injected fault rates to simulate degraded network conditions.

b) Performance Results

Evaluation results demonstrate that the proposed system significantly outperforms all baseline methods across multiple performance dimensions. This section presents detailed findings on latency, throughput, scalability, and resource utilization. For synchronous aggregation latency, the proposed hybrid system achieves a median response time of 78 ms with three concurrent data sources. This represents a substantial improvement over GraphQL Federation (145 ms), the REST Aggregator (210 ms), and the Kafka-centric approach (320 ms), corresponding to reductions of 46%, 63%, and 76%, respectively.

A. Reliability Analysis

Reliability is a critical requirement for production systems that aggregate data from multiple distributed sources, where partial failures are an operational norm rather than an exception. A comprehensive reliability evaluation was conducted across a range of failure scenarios, covering both source-side and infrastructure-level faults. The reliability metric reported in the abstract refers specifically to the proportion of requested data successfully delivered to clients in the presence of source failures. In a controlled experiment, failures were systematically injected into subsets of data sources, and data completeness was measured at the client. Under simultaneous failure of three out of ten sources (a 30% failure rate), the system delivered an average of 99.2% of the requested data in asynchronous mode, demonstrating robust partial-result handling.

B. Case Studies

Case studies were conducted across different application domains to evaluate the practical utility of the proposed system and validate its performance

in real-world integration scenarios. The Smart City Dashboard case study integrated data from traffic sensors, weather services, public transportation APIs, and event calendars to provide a unified operational view of city activity. Performance measurements showed a 40% reduction in dashboard load time compared to the previous approach, which relied on direct and sequential API access. The Healthcare Monitoring case study integrated electronic health records (EHR), real-time patient monitoring devices, prescription databases, and laboratory systems to consolidate comprehensive patient data. Integration using the proposed system required 65% fewer lines of code than the equivalent point-to-point integration approach, significantly reducing development effort and maintenance overhead.

6. Discussion

A. Trade-offs and Design Decisions

The design of the hybrid synchronous-asynchronous API architecture involved deliberate trade-offs aimed at balancing performance, flexibility, reliability, and operational simplicity. Supporting both synchronous and asynchronous interaction patterns introduces architectural complexity, but substantially increases the system's adaptability to diverse client requirements. Synchronous mode is well-suited to interactive applications requiring immediate responses and involves straightforward client-side integration, but constrains processing time and task complexity to maintain acceptable response latency. Asynchronous mode lifts these constraints, enabling more elaborate aggregation pipelines and transformation workflows, at the cost of more sophisticated client logic for job submission, status tracking, and result retrieval. By supporting both patterns within a unified architecture, the system allows applications to select the most appropriate interaction model for each task without committing to a single global strategy. This dual-mode design required careful resource allocation and scheduling. Under concurrent load, synchronous requests are given scheduling priority to preserve low latency for interactive workloads. While this prioritization protects interactive responsiveness, it may delay background asynchronous jobs during periods of high load. The system mitigates this through configurable resource quotas that reserve sufficient capacity for both interaction types, preventing resource starvation while maintaining predictable overall performance. Optimal quota configuration will depend on the workload profile and operational requirements of each deployment.

B. Limitations

While the hybrid API architecture addresses many of the challenges inherent in heterogeneous data integration, several limitations remain that define the boundaries of its applicability and motivate future research. The Redis-based storage model, though highly performant, may encounter constraints with extremely large datasets due to available memory capacity. Although Redis supports persistence mechanisms, storing vast volumes of JSON data entirely in memory may not be cost-effective for all deployment scenarios. Future work could investigate hybrid storage solutions that combine Redis with disk-based systems to achieve more cost-efficient scaling at high data volumes. The system does not perform schema inference, requiring that queries be formulated based on the actual JSON structure of the stored documents. When API response schemas change significantly, existing queries may produce incomplete results or require manual revision. While this design choice reduces ingestion overhead and avoids brittleness associated with inferred schemas, it transfers adaptability responsibility to the query layer. Future enhancements could introduce optional schema versioning or lightweight validation mechanisms to improve query robustness under schema evolution.

C. Comparison with Alternative Approaches

This section provides a detailed comparison of the proposed approach against existing heterogeneous data integration strategies, highlighting relative strengths and limitations across key evaluation dimensions. In terms of data completeness, query flexibility, and development efficiency, the proposed system demonstrates clear advantages over traditional REST aggregation. Conventional REST aggregation requires clients to issue multiple sequential requests and manually join the results, increasing network overhead and client-side complexity. Evaluation results indicate that the proposed approach reduced client-side processing time by 60% relative to this baseline. REST aggregation does, however, offer lower server-side setup cost and may be preferable for simple deployments with a small number of data sources and minimal transformation requirements. GraphQL federation provides a more structured solution by exposing a unified query language and schema across distributed services. Compared to GraphQL, the proposed system offers greater adaptability to heterogeneous and evolving schemas, as it does not depend on predefined schema definitions. Benchmark results show a 35% to 45% reduction in latency for complex aggregation workloads relative to GraphQL federation, attributable primarily to concurrent data fetching and support for partial result delivery.

GraphQL federation, in contrast, offers a more standardized query interface that may be preferable for organizations with existing GraphQL expertise or a preference for declarative query semantics over raw throughput.

D. Future Research Directions

The development and evaluation of the hybrid API system has revealed several promising directions for future research that could advance the broader field of heterogeneous data aggregation. These directions address both current limitations and emerging opportunities in distributed data integration. Edge deployment for IoT gateways represents a natural extension of the proposed architecture, responding to the growing demand for local data processing in resource-constrained IoT environments. Future work could explore lightweight adaptations of the system optimized for edge devices with limited computational capacity and intermittent network connectivity. Such deployments might synchronize selectively with cloud services based on network availability and data criticality, reducing latency for time-sensitive operations while conserving bandwidth. A key research challenge in this context is optimizing Redis-based storage for constrained environments and devising effective synchronization policies that balance local autonomy with global data consistency.

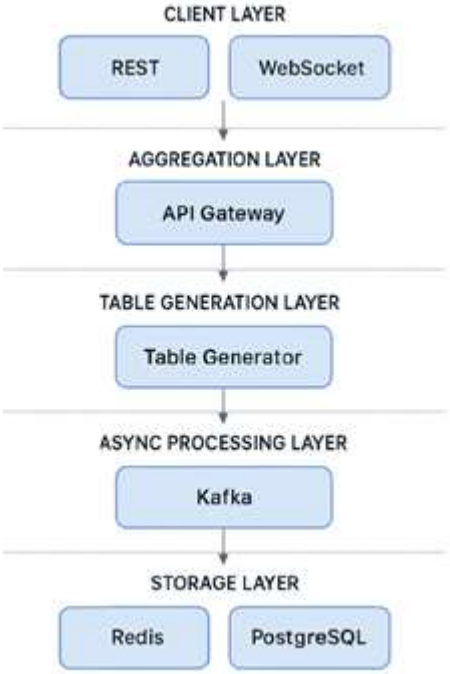

Figure 1. *Layered architecture of the hybrid synchronous-asynchronous API system, showing the five primary layers and their key components.*

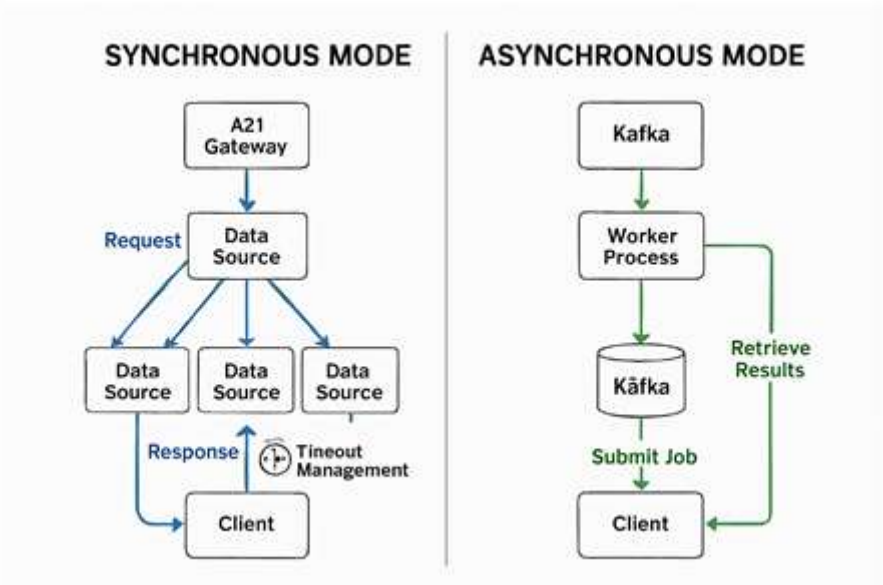

Figure 2. *Comparison of data flow in synchronous mode (left) and asynchronous mode (right), highlighting the key differences in request handling, processing, and response delivery.*

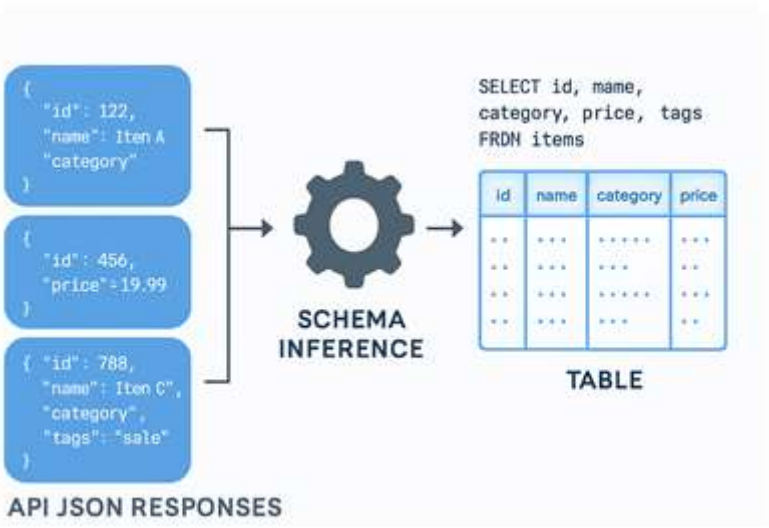

Figure 3. *Message storage and query process in Redis, showing how JSON messages are stored by endpoint with timestamps and queried using a SQL-like interface.*

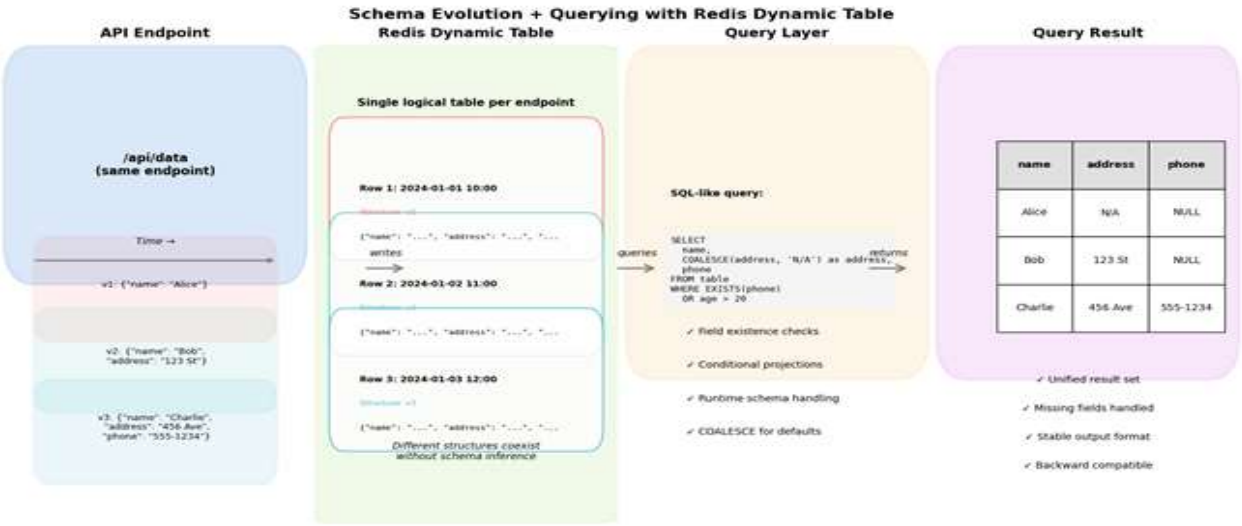

Figure 4. *Schema-tolerant query execution over heterogeneous document collections in Redis, illustrating how the system handles coexisting schema versions from the same API endpoint, with missing fields ignored at query time to preserve backward compatibility.*

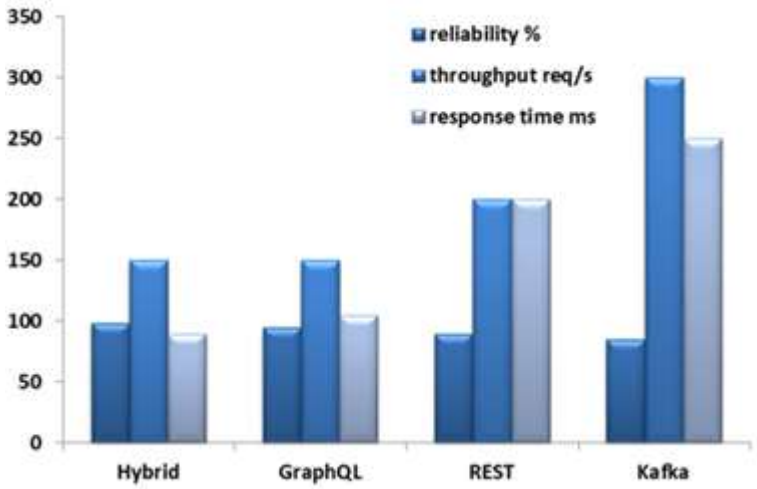

Figure 5. *Performance metrics comparison between our Hybrid API System and alternative approaches (GraphQL Federation, REST Aggregator, and Kafka-centric), showing response time (ms), throughput (req/s), and reliability* (%).

7. Conclusions

This paper has presented a hybrid synchronous-asynchronous API management system capable of dynamic table generation for the aggregation of heterogeneous data. The proposed architecture addresses fundamental limitations of contemporary distributed systems, which increasingly depend on data from diverse sources with heterogeneous schemas and variable availability. Existing approaches such as REST aggregation, GraphQL federation, and standalone event streaming platforms each exhibit significant gaps in addressing the combined requirements of low latency, schema flexibility, fault tolerance, and unified queryability. The proposed system fills these gaps by providing a comprehensive solution that integrates the responsiveness of synchronous processing with the scalability of asynchronous workflows, and that constructs structured, queryable representations from heterogeneous JSON data without requiring predefined schemas or ETL pipelines. The principal contributions of this work are: a dual-mode interaction model enabling clients to select between synchronous and asynchronous patterns based on task requirements; automatic dynamic table generation from heterogeneous API responses stored in Redis with reception timestamps; elimination of schema inference by storing JSON messages in their native structure; and a complete reference implementation integrating these capabilities within a unified architecture. Together, these contributions address the core challenges of schema heterogeneity, latency variability, partial source failures, and persistent queryable storage that have historically complicated multi-source data aggregation. The results of this work carry implications beyond the specific system described. They demonstrate that synchronous and asynchronous processing paradigms can be unified within a single coherent architecture, challenging the conventional assumption that these patterns require separate infrastructures. The Redis-based storage approach offers a novel foundation for schema-agnostic data integration that minimizes manual configuration and adapts naturally to structural evolution without inference-based preprocessing. The capacity for on-the-fly table generation bridges the gap between API consumption and database functionality, pointing toward a more unified model of data management in distributed systems.

Several directions for future research emerge from this work. Edge deployment for IoT gateways would extend the architecture to resource-constrained environments where connectivity is intermittent and local processing is essential. Integration with federated learning frameworks could leverage the system's data harmonization capabilities to support distributed machine learning without centralizing raw data. The development of ethical frameworks for multi-source aggregation could help identify and mitigate privacy or bias concerns arising from the combination of heterogeneous data streams. As distributed systems continue to proliferate and the volume and variety of available data sources grows, the demand for robust integration infrastructure will only increase. The proposed hybrid API architecture constitutes a meaningful step toward meeting this demand, offering a flexible, efficient, and reliable platform for the next generation of data-intensive applications. By synthesizing the strengths of prior approaches and introducing new capabilities for schema-agnostic dynamic table generation, this work contributes both practical tooling and conceptual advances to the field of heterogeneous data aggregation.

Author Statements:

- **Ethical approval:** The conducted research is not related to either human or animal use.
- **Conflict of interest:** The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper
- **Acknowledgement:** The authors declare that they have nobody or no-company to acknowledge.
- **Author contributions:** Conceptualization, H. Chaalal; methodology, H. Chaalal, F. Khatemi, and M.A. Chemrak; software, H. Chaalal; validation, H. Chaalal, F. Khatemi and M.A. Chemrak; formal analysis, H. Chaalal and F. Khatemi; investigation, H. Chaalal and M.A. Chemrak; resources, H. Chaalal and F. Khatemi; data curation, H. Chaalal and M.A. Chemrak; writing---original draft preparation, H. Chaalal; writing---review and editing, H. Chaalal and F. Khatemi; visualization, H. Chaalal and M.A. Chemrak; supervision, H. Chaalal.
- **Funding information:** The authors declare that there is no funding to be acknowledged.
- **Data availability statement:** The data that support the findings of this study are available on request from the corresponding author. The data are not publicly available due to privacy or ethical restrictions.
- **Use of AI Tools:** The author(s) declare that no generative AI or AI-assisted technologies were used in the writing process of this manuscript.

References

[1] M. A. Baazizi, H. B. Lahmar, D. Colazzo, G. Ghelli, and C. Sartiani, "Schema inference for massive JSON datasets," in *Extending Database Technology (EDBT)*, 2017, pp. 222-233.

[2] K. Banker, *MongoDB in Action*. Manning Publications, 2016.

[3] L. Byron, "GraphQL: A data query language," Facebook Engineering Blog, Sep. 2016. [Online]. Available: https://engineering.fb.com/2016/09/12/data/graphql-a-data-query-language/

[4] R. Cappuzzo, P. Papotti, and S. Thirumuruganathan, "Creating embeddings of heterogeneous relational datasets for data integration tasks," in *Proc. ACM SIGMOD Int. Conf. Management of Data*, 2020, pp. 1335-1349.

[5] T. Dunning and E. Friedman, *Streaming Architecture: New Designs Using Apache Kafka and MapR Streams*. O'Reilly Media, 2016.

[6] GraphQL Foundation, "GraphQL Specification," June 2018. [Online]. Available: https://graphql.org/

[7] R. T. Fielding, "Architectural styles and the design of network-based software architectures," Ph.D. dissertation, University of California, Irvine, 2000.

[8] M. Fowler, "BFF: Backend for Frontend," martinfowler.com, 2015. [Online]. Available: https://martinfowler.com/articles/backend-for-frontend.html

[9] E. Gallinucci, M. Golfarelli, and S. Rizzi, "Schema profiling of document-oriented databases," *Information Systems*, vol. 75, pp. 13-25, 2018.

[10] E. Huang, L. Xu, and T. Zhang, "TiDB: A Raft-based HTAP database," *Proc. VLDB Endowment*, vol. 13, no. 12, pp. 3072-3084, 2020.

[11] S. Idreos, K. Deng, T. Kraska, S. Madden, M. Stonebraker, and J. Yang, "The Data Civilizer system," in *CIDR*, 2019.

[12] K. Indrasiri and P. Siriwardena, *Microservices for the Enterprise: Designing, Developing, and Deploying*. Apress, 2021.

[13] J. Kreps, "Questioning the Lambda Architecture," *O'Reilly Radar*, 2014. [Online]. Available: https://www.oreilly.com/radar/questioning-the-lambda-architecture/

[14] J. Kreps, N. Narkhede, and J. Rao, "Kafka: A distributed messaging system for log processing," in *Proc. NetDB*, 2011, pp. 1-7.

[15] N. Marz and J. Warren, *Big Data: Principles and Best Practices of Scalable Realtime Data Systems*. Manning Publications, 2015.

[16] B. M. Michelson, "Event-driven architecture overview," *Patricia Seybold Group*, vol. 2, no. 12, pp. 10-1571, 2006.

[17] S. Mudgal, H. Li, T. Rekatsinas, A. Doan, Y. Park, G. Krishnan, and F. Naumann, "Deep learning for entity matching: A design space exploration," in *Proc. ACM SIGMOD Int. Conf. Management of Data*, 2018, pp. 19-34.

[18] S. Newman, *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media, 2015.

[19] S. Newman, *Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith*. O'Reilly Media, 2019.

[20] F. Özcan, Y. Tian, and P. Tözün, "Hybrid transactional/analytical processing: A survey," in *Proc. ACM SIGMOD Int. Conf. Management of Data*, 2017, pp. 1771-1775.

[21] A. Pavlo, G. Angulo, J. Arulraj, H. Lin, J. Lin, L. Ma, and M. Stonebraker, "Self-driving database management systems," in *CIDR*, 2017.

[22] E. Rahm and P. A. Bernstein, "A survey of approaches to automatic schema matching," *The VLDB Journal*, vol. 10, no. 4, pp. 334-350, 2001.

[23] C. Richardson, *Microservices Patterns: With Examples in Java*. Manning Publications, 2019.

[24] N. Shamgunov, "The MemSQL database: Technology and performance," in *Proc. ACM SIGMOD Int. Conf. Management of Data*, 2017, pp. 1737-1738.

[25] J. Strachan, "JSONata: JSON query and transformation language," Technical Report, 2017. [Online]. Available: https://jsonata.org/

[26] A. Vázquez-Ingelmo, J. Cruz-Benito, and F. J. García-Peñalvo, "Improving the OEEU's data-driven technological ecosystem's interoperability with GraphQL," in *Proc. 7th Int. Conf. Technological Ecosystems for Enhancing Multiculturality*, 2020, pp. 1-8.

[27] E. Wittern, A. Cha, J. C. Davis, G. Baudart, and L. Mandel, "An empirical study of GraphQL schemas," in *Int. Conf. Service-Oriented Computing*, 2019, pp. 3-19.

[28] G. Young, "CQRS Documents," Technical Report, 2010. [Online]. Available: https://cqrs.files.wordpress.com/2010/11/cqrs_documents.pdf

[29] A. Caraffa, "Synchronous and Asynchronous APIs," OpenAPI Blog, Dec. 2025. [Online]. Available: https://openapi.com/blog/synchronous-and-asynchronous-apis

[30] Gravitee.io, "Gravitee: API Management Platform for APIs, Events & Agents," 2025. [Online]. Available: https://www.gravitee.io/

[31] Nordic APIs, "Top 10 API Gateways in 2025," Nordic APIs, Jun. 2025. [Online]. Available: https://nordicapis.com/top-10-api-gateways-in-2025

[32] L.Cogan et al, Redis, "Redis 8.2 GA: Performance, Efficiency, and New Commands," Redis Blog, Aug. 2025. [Online]. Available: https://redis.io/blog/redis-82-ga

[33] L. Lakshika, *et al.* "Async Messaging at Scale: Kafka + Redis Streams + Spring Boot (20,000 RPS Blueprint — 2025 Edition)," Stackademic, Nov. 2025. [Online]. Available: https://blog.stackademic.com/async-messaging-at-scale-kafka-redis-streams-spring-boot-20-000-rps-blueprint-2025-7d19a3f1f746

[34] C. Reddy Kasaram, *et al.* "Harnessing Asynchronous Patterns with Event Driven Kafka and Microservices Architectures," ResearchGate, Oct. 2023 (published

version 2025). [Online]. Available: https://www.researchgate.net/publication/397530105

[35] A. Jain, *et al.*"Building a High-Frequency Trading System With Hybrid Strategy (Redis & InfluxDB): From 10ms to Sub-Millisecond Latency," Medium, Nov. 2025. [Online]. Available: https://vardhmanandroid2015.medium.com/building-a-high-frequency-trading-system-with-hybrid-strategy-redis-influxdb-from-10ms-to-85716febefcb

[36] J. Boyer, *et al.* "Reference Architecture - Event-Driven Solutions in Hybrid Cloud," GitHub Pages, 2025 (ongoing). [Online]. Available: https://jbcodeforce.github.io/eda-studies/concepts/eda

[37] *Koukaras, Paraskevas et al. "Data Integration and Storage Strategies in Heterogeneous Analytical Systems: Architectures, Methods, and Interoperability Challenges." Inf. 16 (2025): 932.*

[38] *Sserujongi, Richard et al. "Design and Evaluation of a Scalable Data Pipeline for AI-Driven Air Quality Monitoring in Low-Resource Settings." ArXiv abs/2508.14451 (2025): n. pag.*

[39] Arafat, Jahidul et al. "Next-Generation Event-Driven Architectures: Performance, Scalability, and Intelligent Orchestration Across Messaging Frameworks." ArXiv abs/2510.04404 (2025): n. pag.