UDC 667.021.1 DOI https://doi.org/10.35546/kntu2078-4481.2025.2.2.15

O. S. DEMIDOV
Postgraduate Student at the Department of Specialized Computer Systems
National University "Lviv Polytechnic"
ORCID: 0009-0004-3464-1655

O. Y. HONSOR
Candidate of Technical Sciences,
Associate Professor at the Department of Specialized Computer Systems
National University "Lviv Polytechnic"
ORCID: 0000-0003-0895-5859

# CONFIGURING THE STRUCTURE OF THE SERVERLESS SYSTEM FOR EFFICIENT DATA COLLECTION

*Collecting and analyzing the vast amount of data in different formats and from different vendors was always important for modern software of all kinds. It brings us to the urge of creating a system that is flexible, scalable, and effective with capabilities such as fetching, processing, and storing this data. Modular abstraction helps in making the system generic and overcoming some limitations of data providers and vendors. Also helping to handle different formats and process high-volume workloads in parallel with a comprehensive error-handling strategy. Serverless architecture also plays a huge part in achieving desired results. The essential architectural elements are: AWS Lambda, S3, Event Bridge, SQS, and Athena. All of them are the core of building a fault-tolerant, efficient pipeline which can work with various domains, API's and handle different regulations. Metadata-driven orchestration approach allows seamless vendor integration and support. Moreover, those methods allows the system to be more trackable, easy to test and secure, due to basic cloud solutions provided by certain cloud-provider, which is being used. Modular experimental estimation demonstrates the effectiveness of the system compared to alternative solutions that are currently widely used. It's a new step for the collection of data from 3rd party resources as it's also providing more versatility and configurability on top of discussed topics.*

*This research is focused on the creation of an advanced pipeline for analyzing climate and environment-related datasets from various external providers (collection, organization, storage, transformation, analysis). The solution must be flexible and adapt to different user needs and extensions. It shouldn't only focus on the initial state of the data collection process, but extend beyond that, to be prepared to effectively operate data with other external services. The system should be ready to handle a list of factors that may appear during the process of data collection. Such as: non-consistent data/ metadata, quotas, server downtime, dynamic schema evolution, and so on. For these reasons, it is advisable to propose a modular, event-driven, and serverless architecture that will lead to the orchestration of the entire workflow described.*

*The system should allow the dynamic adjustments to data-fetching strategies based on relevant situation, real-time statistics and providers reliability. The architecture supports extensibility for future data sources and analytical modules, promoting long-term maintainability and adaptability. Integration with services like AWS Secrets Manager and MongoDB enhances security and centralized management of access credentials and vendor state. Furthermore, the use of Athena-based querying enables near real-time analytical capabilities over large, semi-structured datasets, supporting advanced insights generation. These design decisions contribute to a resilient, future-proof solution tailored to the needs of complex, large-scale data ecosystems.*

***Key words:*** *Data-collecting, Serverless Architecture, AWS, Cloud data pipelines, Environmental data.*

О. С. ДЕМІДОВ
аспірант кафедри спеціалізованих комп'ютерних систем
Національний університет «Львівська політехніка»
ORCID: 0009-0004-3464-1655

О. Й. ГОНСЬОР
кандидат технічних наук,
доцент кафедри спеціалізованих комп'ютерних систем
Національний університет «Львівська політехніка»
ORCID: 0000-0003-0895-5859

## НАЛАШТУВАННЯ СТРУКТУРИ БЕЗСЕРВЕРНОЇ СИСТЕМИ ДЛЯ ЕФЕКТИВНОГО ЗБОРУ ДАНИХ

*Збір та аналіз величезної кількості даних у різних форматах та від різних постачальників завжди був важливим для сучасного програмного забезпечення всіх видів. Це спонукає нас до прагнення створити гнучку, масштабовану та ефективну систему з такими можливостями, як вибірка, обробка та зберігання цих даних. Модульна*

*абстракція допомагає зробити систему універсальною та подолати деякі обмеження постачальників даних та постачальників. Також допомагає обробляти різні формати та великі обсяги робочих навантажень паралельно з комплексною стратегією обробки помилок. Безсерверна архітектура також відіграє величезну роль у досягненні бажаних результатів. Основними архітектурними елементами є: AWS Lambda, S3, Event Bridge, SQS та Athena.*

*Всі вони є основою побудови відмовостійкого, ефективного конвеєра, який може працювати з різними доменами, API та обробляти різні правила. Підхід оркестрації на основі метаданих забезпечує безперешкодну інтеграцію та підтримку постачальників. Крім того, ці методи дозволяють системі бути більш відстежуваною, легкою в тестуванні та безпечною завдяки базовим хмарним рішенням, що надаються певним хмарним постачальником, який використовується. Модульна експериментальна оцінка демонструє ефективність системи порівняно з альтернативними рішеннями, які зараз широко використовуються. Це новий крок у зборі даних зі сторонніх ресурсів, оскільки він також забезпечує більшу гнучкість та налаштування на додаток до обговорюваних тем.*

*Це дослідження зосереджено на створенні вдосконаленого конвеєра для аналізу наборів даних, пов'язаних з кліматом та навколишнім середовищем, від різних зовнішніх постачальників (збір, організація, зберігання, перетворення, аналіз). Рішення має бути гнучким та адаптуватися до різних потреб користувачів та розширень. Воно повинно зосереджуватися не лише на початковому стані процесу збору даних, але й виходити за його межі, щоб бути готовим до ефективної роботи з даними з іншими зовнішніми сервісами. Система повинна бути готова обробляти перелік факторів, які можуть виникнути під час процесу збору даних. Такі як: неузгоджені дані/метадані, квоти, час простою сервера, динамічна еволюція схеми тощо. З цих причин доцільно запропонувати модульну, керовану подіями та безсерверну архітектуру, яка призведе до оркестрації всього описаного робочого процесу.*

*Система повинна дозволяти динамічне налаштування стратегій отримання даних на основі відповідної ситуації, статистики в режимі реального часу та надійності постачальників. Архітектура підтримує розширюваність для майбутніх джерел даних та аналітичних модулів, що сприяє довгостроковій підтримці та адаптивності. Інтеграція з такими сервісами, як AWS Secrets Manager та MongoDB, підвищує безпеку та централізоване управління обліковими даними доступу та станом постачальників. Крім того, використання запитів на основі Athena забезпечує аналітичні можливості майже в режимі реального часу для великих, напівструктурованих наборів даних, підтримуючи генерацію розширених аналітичних даних. Ці проектні рішення сприяють створенню стійкого, перспективного рішення, адаптованого до потреб складних, масштабних екосистем даних.*

***Ключові слова:*** *Збір даних, Безсерверна архітектура, AWS, Хмарні конвеєри даних, Дані про навколишнє середовище.*

## Problem statement

Large-scale data collection systems with the ability to adapt to different data sources at any complexity, data structure, scale, and do it efficiently have never been in such demand as it is right now. In particular, any domains like Artificial Intelligence, finance, IOT, scientific observations, and monitoring highly require such a system. I would even say that they are very limited without it and have a massive disadvantage compared to those who have one. As a researcher who tries to develop general architecture solutions as well as structure and algorithms for such systems, I needed to find a dataset to test my system to its full potential. And I think there is no better field for it than Earth observation, climate monitoring, and analysis. Those analytics require a vast amount of highly heterogeneous data. From satellite imagery, weather observations, and sensor readings. Those data points are sourced from different academic repositories, corresponding vendor APIs, and agencies that work in this field and expose their observations.

## Analysis of Recent Research and Publications

Fortunately, a number of professional agencies are willing to make their data available free of charge. Examples include NASA, NOAA and ESA, as well as many others. They created an unprecedented ability for scientists to have a deep look, capture their data, and even create a predictive model based on the received data. Despite the vast amount of available data, the technical challenge lies in receiving, transforming, and storing this data, moreover, doing it in a scalable, efficient, and cost-effective manner. Those datasets have different vendors and differ not only in base protocols (like FTP, images, REST APIs), but also in their structure (JSON, binary, CSV, XML, even multi-gigabyte archives). They are updated frequently, and to really keep track of it, we need a real-time data collection and analysis. Another issue that we are dealing with is the requirement for robust standardisation and metadata. This is an essential requirement for the uniform querying of data and its processing via machine learning models [1].

Traditional ETL (Extract, Transform, Load) pipelines mostly lack the scalability, cost-effectiveness, and flexibility to handle different data problems that might appear. Typically, they depend on central orchestration and manual handling of some specific cases [2]. Also, maintaining such architecture becomes harder and harder with new data vendors. Modern serverless solutions are handling reactive data pipelines a lot better and with the correct setup of the system, can scale elastically, execute specific logic in isolated units to reduce overhead by handling infrastructure management to cloud providers.

AWS cloud services, including Lambda, S3, EventBridge, and Athena, will serve as the foundation for the prototype system. Those are so-called general building blocks that will always exist, as they are essential for building such a solution. They are facilitated by decoupled computing, flexible storage, and dynamic changes in orchestration of the

system. Amazon Athena and Sagemaker are the tools that help the system to prepare data in dynamic formats, structure it, so that we can delegate it to machine learning pipelines for analysis [3].

**Formulation of Research Objective**

This article represents a proposed architecture and structure solution, as well as an estimate of the efficiency of a cloud-native system tailored for data collection and processing. It's based on the prototype of such a system in the environmental and climate change domain.

Key features are:

Event-driven orchestration. All data from its start (database with metadata) all the way to a fully processed and structured model will go through a loosely coupled system with services such as EventBridge, SQS, Lambda and Athena

Diverse data formats support. Intelligent handling of various payloads with different types of responses and sizes

Standardization and transformation workflows. Processing various data into a single acceptable model with a unified structure that is compatible with Athena and downstream applications

Flexible and extendable vendor data integrations. Allow new vendors to be added anytime and have 0 downtime for them as soon as they are added. Create real real-time collection/processing system.

This work proposes a practical plan for future investigations and improvements both in general and specifically in a specified domain of science. Introducing scalable data ingestion for Earth system science and cyber-physical systems.

**Presentation of The Main Research Material**

To address challenges mentioned in the introduction of this article, namely: scalability, elasticity, flexibility, fault tolerance, data ingestion from diverse sources and formats, and also cost efficiency, the system is built in a very modular and loosely coupled cloud-native manner. It's not relying on monolithic, batch-oriented solutions nor on hardware dependencies of a static on-premise approach. It embraces isolation and hardware independence, being fully event-driven and fault-tolerant. This will open the opportunities for isolation of logic from different vendors by the layer of abstraction and result in high flexibility, parallelism, and automation.

The architecture consists of a few independent layers. Each of them serves as a solution to some task in the life cycle of data. Starting from planning, metadata preparation, and orchestration, or processing and data fetching requests [1]. This multi-layer model improves fault-tolerance, traceability, and enables easy integration of new data vendors and formats.

So there are 5 main layers. If necessary, the addition of extra layers is permitted. However, for the prototype of the systems, these layers are sufficient.

Scheduler Layer. As stated, it's used to schedule jobs. This schedule might depend on multiple factors, but initially it's set by configurable cron expressions or event-based triggers (coming from the database with all vendors). But scheduling can also happen due to job failure or unavailability of the server at some point in time (so-called retrials).

Orchestration Layer. This is the first place where the scheduled request comes. Coordination of request, validation, and circuit breaker logic of handling errors happens here.

Execution Layer. Executes specific vendor commands using modular Lambda functions (orchestrated by SQS queue). It's responsible for fetching data, processing, and storing it into general S3 storage.

Data Layer. Persists clean data in S3, then aligns it with the desired model by Athena and stores it again. Metadata is managed in a NoSQL database (MongoDB) separately [6].

Analytics layer. This layer is flexible, but mainly for exposing collected and structured datasets to machine learning pipelines (such as Sagemaker). Enabling efficient ad-hoc queries.

The core part of this architecture lies in the fact that those layers don't intersect with each other [4]. Allowing further improvement, redesign, or domain or requirements change of the system [5].

**Vendor Abstraction Layer**

Being able to communicate with a wide range of data sources with unique data formats, API specifications, rate and speed limits, and authentication mechanisms is a crucial part of the system with such obligations. The vendor abstraction layer was designed to address those challenges, but also avoid tight coupling and introduce additional complexity. This layer isolates vendor-specific logic that allows for extending the system without any downtime or additional challenges [7].

Each vendor is a dynamic module that relies on abstraction. Its abilities are defined by metadata, stored in MongoDB. Those configurations include (but are ot limited to): API endpoints, request rate limits, retrial state and policies, secret name references for authentication, expected data formats, scheduling, pagination, API specific information for the desired data response [8].

At runtime, those parameters are interpreted by the system to define which data to fetch, how often, retrial policies, and so on. Implementation of this layer uses the Command pattern. Each operation receives data like a data object, which encapsulates the logic of communication with the server, data processing, and storage. Mostly, vendors consist of a collector – here lies everything related to getting data, paginating, and retrying. The repository component is responsible for parsing, transforming, compressing data, and writing it to appropriate S3 buckets in specified format and size batches.

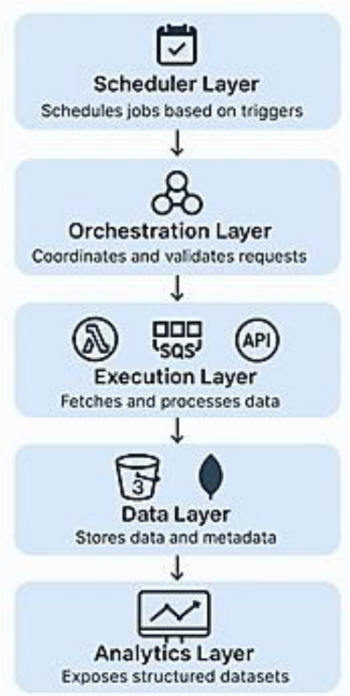


**Fig. 1. Architecture layers**

**Parallel Execution and Observability**

One of the advantages of the proposed architecture is independent command execution from the vendor. Each of them is being processed by different AWS Lambda functions. Those commands are coming in parallel using Amazon SQS. This construction allows for building an independent and scalable operating part that adapts to different spikes in workload. Basically, it removes the bottleneck from the whole fetching process.

It increases productivity, as well as traceability. We can use each message and identify which of them failed and why. Also, retrying them is always an option. Each command can be tested independently and in isolation from other parts of the system, which makes it more robust and easier to maintain in terms of testing. Also, security benefits from it as each execution is bound to a specific vendor and its related credentials and can't change any other data.

**Error Handling, Retries, and Resilience Mechanisms**

So, to adapt to different issues with data fetching, a variety of methods are provided in the system. Starting from default error handling and logging to the DLQ (dead letter queue – re-triggers failed SQS messages), circuit breaker (to handle exceeding rate limitations and server downtime) and others.

Each vendor endpoint is actively monitored by its recent behavior. If there are a few repetitive errors or time-outs, malformed payloads or throttling responses, system transitions failing vendor (or its part) to the Opened state. This means that all requests to this route will be stopped. That will protect both our system and vendor, not allowing them to overload with a vast amount of retry requests or corrupted data. After the predefined cooldown period goes by, the system transitions this vendor to the Half-open state. Which means that now it will try to make a couple of requests to verify that the server is responding correctly. If that's the case, the circuit breaker will switch to Closed. And it's the default state for the system. This indicates that all components are functioning optimally, allowing the system to utilize the server's full capacity.

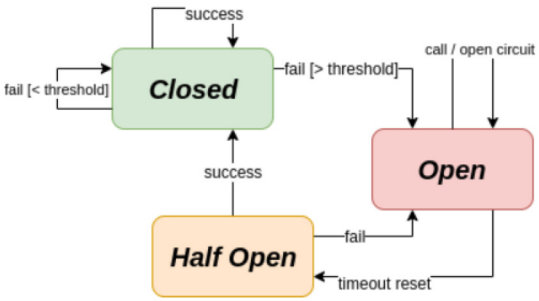


**Fig. 2. Circuit breaker process**

Additionally, the application is designed to adhere to a retry policy that is entirely independent of the circuit breaker itself. We got retries on the request level as well as the orchestration level. They both use an exponential backoff pattern. Those methods allow the system to handle all failure points. Both server-side and user-side failures are like human mistakes, while setting up the system (with invalid timeout or credentials). Allowing fine-grained control on the number of retries, which should be attempted before we escalate errors to handle them by manual resolution (with logs or notifications). But most of the time, given that vendors work correctly, the system utilizes self-recovery in the face of partial failure or degraded conditions [9].

**Experimental Evaluation**

To validate the efficiency and adaptability of the proposed architecture, we will create the system and simulate a representative use case. We will be measuring the most relevant metrics for this challenge [10, 11]. We are going to collect data from a list of global providers' heterogeneous datasets to analyze the process of their collection, processing, transforming, and storing. These sources cover various domains and data types. Our database includes simple JSON files as well as atmospheric conditions and satellite images. The selected data sources forthe prototype system are:

NASA GES DISC: providing datasets available via REST and FTP technologies in NetCDF and CSV formats

ESA Copernicus Open Access Hub. Offers satellite data in GeoJSON format

OpenAQ. JSON based public API for real-time air quality

Simulated Retail Data Vendor. E-commerce reporting API with JSON data, rate-limits and token-based access.

Each vendor is integrated as a separate dynamic module using metadata stored in MongoDB. Their logic is based on abstraction, so we can easily switch those resources if needed (as long as they contain abstraction that we already manage (types, formats). After the initial test run, distinct collection jobs were executed automatically, triggered by scheduled retry-based events. During the test run system experienced some rate limitations and errors, but handled it well, sustaining an average concurrency of 15–20 Lambda functions.

Data was successfully fetched in batches if that was possible. Then normalized to a JSON general structure. Then it was compressed and stored in S3. Metadata for the corresponding fetch updated the MongoDB database. For most of the data, we received transformed structured analytics using Amazon Athena, and it would be ready to be sent to Sagemaker for further processing.

Productivity of the system was evaluated by the few metrics. Here are results that represent average values during the experimental run.

Cold starts of the system were only observed when no activity was present for last moments before it, due to backoffs or just free time between scheduled tasks [12]. And lasted less than half a second. So the more system is loaded the better in terms of having less cold start time. Most of the tasks are taking only couple of seconds, but it depends on provider. This is good approach, to have small data batches and can be achieved by using pagination for most cases.

Table 1

**System metrics evaluation**

| Metric | Proposed Serverless System |
|---|---|
| Throughput | ~4.3 MB/sec (aggregated) |
| Cold Start Time | ~280 ms (Lambda, Node.js) |
| Execution Duration | ~2.1 seconds (average job) |
| API Error Rate | 0.9 % (mostly 429/503 errors) |
| Retries (successful) | 97 % recovery within 2 tries |
| S3 Compression Efficiency | ~78 % (GZIP, structured JSON) |
| Cost per 1000 Jobs | ~$0.84 (all-inclusive) |

Fault tolerance was highly effective as well. Most of API errors were caused by throttling and processed well with retrials and time. Circuit breaker mechanism also prevented cascading failures by pausing access to unavailable endpoints.

Archieved data appeared to be effective and took less space than JSON raw data by 78 %. Hovewer, not all data can be shrunk by that volume, it depends on specific data types.

At last, cost profile appeared to be as expected, but also can be reduces if scale of the system goes up. Most expences were caused by usage on Lambda functions and S3 load.

For a better understanding of system effectiveness and the real advantages of it, the system will be compared with common alternatives, which are commonly used in data integration systems and their automation.

Monolithic ETL systems (e.g, Airflow on EC2): tightly coupled, synchronous, often requiring static resource provisioning

Batch-oriented EC2 jobs. They are more elastic, but require a lot of effort to set them in a fault-tolerant way. They are also less agile.

Manual ingestion workflows. Most widely used legacy solution. Based on scripts, cron jobs, and CLI tools. Non-cloud solution.

Table 2

**Solutions comparison**

| Metric | Serverless (Ours) | Monolithic ETL | EC2 Batch Jobs | Manual Ingestion |
|---|---|---|---|---|
| Scalability | High | Low–Medium | Medium | Low |
| Latency (Avg. Job) | 2.1 sec | 5–12 sec | 8–15 sec | Min–Hours |
| Fault Isolation | Strong | Weak | Moderate | None |
| Ops Overhead | Very Low | High | High | Very High |
| Cost (per 1000 jobs) | $0.84 | $3.10 | $2.75 | N/A |
| Cold Start Risk | Moderate | N/A | Low | N/A |
| Retry/Recovery | Fully automated | Partial/manual | Manual | Manual |

As we see from the comparison table, the serverless model is clearly having advantages in terms of resilience, elasticity, and cost-efficiency. Monolithic ETL tools allow having powerful orchestration features, but require manual error handling and recovery [13]. Also addition of new vendors and data types requires centralized configuration and a system restart. EC2 solutions allow horizontal scaling, but resource management and error-handling, and vendor isolation remain a challenge.

At the end of the day, proposed system not only winning in most of the metrics, it also is modern-cloud oriented solution. It's fairly easy to migrate to it and also it rapidly evolves. It embrases decoupling, automation and granular observability. And those plays a big role for sustainability of large system with vast amount of data [14].

## Limitations and Future Directions

While the proposed system performs well and offers great conditions for collecting, processing, and storing data in various cases and complexities, a few areas are still widely open for improvements and research. First of all, there are several topics to consider, including handling greater scale, managing larger workloads (such as substantial data processing required in the initial fetch phase), streaming data, intelligent semantic alignment, and more. Furthermore, adapting the system to achieve the desired result is essential, as modifications are encouraged in the event of specific circumstances at any point in the data pipeline.

One of the primary restrictions in the cloud is the reliance on batch-based data ingestion. Although the system has performed adequately in retrieving data at scheduled intervals, it does exhibit delays when processing data streams, event-driven data, or real-time sensor streams. Future development should be focused on the integration of AWS Kinesis data streams along with AWS MSK (Managed Kafka). This can be implemented as an optional add-on to the core architecture, provided that the aforementioned needs are met. These will help support stream-based architecture while preserving modular structure. However, implementing this change would necessitate not only architectural enhancements but also structural modifications to the codebase. One area that could benefit from attention is the enhancement of collector and repository logic to operate on event- or data-driven foundations and respond to incoming batches [15].

Another key challenge is the normalization of the scheme between the system and a vendor. Presently, the predominant data format is JSON, as it is the most widely used data format for non-strict schemes. However, it does not resolve semantic differences in field names, units, or structures for equivalent concepts. Addressing this may involve integrating AI and providing training. Automation has the potential to introduce complexity, but it also ensures the system's resilience to conversion of inches to centimeters during flight operations demonstrates this capability [16].

The proposed architecture is well-positioned to support machine-learning processes. This pertains to individuals involved in Sagemaker-related projects. The article addresses this topic briefly; nevertheless, it is a crucial element, as it determines the system's primary function. However, the data is already structured and prepared to go into Sagemaker models. Therefore, as a future improvement, it is advisable to implement an automated machine learning process as part of the data pipeline. This will finalize the cycle between data acquisition and its direct outcome, which will be analyzed data, results, or trained models [17].

Lastly, for most advanced use cases, when data volumes reach the petabyte scale, there can be natural limitations by the services such as Athena and S3. It will occur due to query latency and partition depth. It can be prevented by future integrations with Lakehouse (AWS Iceberg), which can combine the flexibility of cloud storage with the consistency of relational stores.

Those directions point out weak places in the solution. Even though they are specific, and proposed system can handle most of the needs. This brings us to a broader vision, where modular architecture not only supports fetching, fault-tolerance, processing, storing, and analyzing data, but also the full data lifecycle of scientific insight.

## Summary and Conclusion

This paper presents a novel, modular, and serverless system architecture for efficient and scalable data collection, transformation, and processing, with a particular focus on climate and environmental datasets. The proposed architecture effectively addresses the critical challenges faced by traditional ETL systems, such as limited scalability, fault tolerance, and high operational overhead.

Through the integration of AWS cloud services, inluding Lambda, S3, EventBridge, SQS, and Athena, the system achieves elasticity, high throughput, automated error recovery, and cost-efficiency. Experimental evaluation demonstrates that the serverless model outperforms monolithic and batch-based alternatives in terms of latency, scalability, reliability, and operational simplicity. The system's vendor abstraction layer further enhances flexibility, allowing seamless integration of heterogeneous data providers with minimal manual intervention.

The inclusion of parallel processing, circuit breaker logic, and fine-grained retry mechanisms significantly increases fault tolerance and enables real-time adaptability to diverse vendor constraints and API behaviors. Additionally, the modular design allows the architecture to evolve and scale as new requirements or data sources emerge.

Despite its strengths, the current implementation faces limitations in handling real-time streaming data and semantic heterogeneity in data formats. Future research directions include the incorporation of streaming services like AWS Kinesis and Kafka, the automation of schema normalization using AI techniques, and the integration of machine learning pipelines for intelligent data analysis.

In conclusion, this research offers a practical and forward-looking solution for scientific and industrial domains that demand flexible, fault-tolerant, and efficient data ingestion infrastructures. The proposed system lays the foundation for a comprehensive, cloud-native platform capable of supporting the full lifecycle of data-driven insight.

**Bibliography**

1. Балдіні, Іоана та ін. «Безсерверні обчислення: сучасні тенденції та відкриті проблеми». Досягнення досліджень у хмарних обчисленнях (2017): 1–20. https://doi.org/10.1007/978-981-10-5026-8_1

2. Спіллнер, Йозеф та ін. «FaaSdom: набір еталонів для безсерверних обчислень». Міжнародна конференція IEEE/ACM з комунальних послуг та хмарних обчислень (UCC) 2019 року. IEEE, 2019. https://doi.org/10.1109/UCC48980.2019.00015

3. Йонас, Ерік та ін. «Спрощене хмарне програмування: погляд Берклі на безсерверні обчислення». Препринт arXiv arXiv:1902.03383 (2019). https://doi.org/10.48550/arXiv.1902.03383

4. Ван, Лей та ін. «Зазирнувши за лаштунки безсерверних платформ». Матеріали щорічної технічної конференції USENIX 2018 року (USENIX ATC 18). 2018. https://doi.org/10.5555/3291168.3291184

5. Ройтер, Крістіан та ін. «Підхід на основі шаблонів до стійкості до помилок API для хмарних систем». Матеріали Міжнародної конференції IEEE з хмарної інженерії (IC2E) 2020 року. IEEE, 2020. https://doi.org/10.1109/IC2E48795.2020.00011

6. Ван Ейк, Елко та ін. «Еталонна архітектура SPEC-RG для FaaS: від мікросервісів та контейнерів до безсерверних платформ». IEEE Internet Computing, т. 23, № 6, 2019, с. 7–18. https://doi.org/10.1109/MIC.2019.2952061

7. Річардсон, Кріс. Шаблони мікросервісів: з прикладами на Java. Manning Publications, 2018. https://doi.org/10.1007/978-1-4842-5574-3_3

8. Медель, Віктор та ін. «Підтримка проміжного програмного забезпечення для контекстно-залежних застосунків з компонентами повторного використання». Journal of Systems and Software 134 (2017): 181–197. https://doi.org/10.1016/j.jss.2017.08.009

9. Резенде, Педро та ін. «Відмовостійка архітектура для API на основі REST». Future Generation Computer Systems 108 (2020): 422–435. https://doi.org/10.1016/j.future.2020.03.005

10. Спіллнер, Йорг та Мірко Вестерхайде «Безсерверна обробка даних за допомогою AWS Lambda та Apache Spark». Journal of Cloud Computing, т. 9, № 1, 2020, с. 1–14. https://doi.org/10.1186/s13677-020-00172-0

11. Балдіні, Іоана та ін. «Безсерверні обчислення: сучасні тенденції та відкриті проблеми». Research Advances in Cloud Computing, під редакцією Раджкумара Буйї та ін., Springer, 2017, с. 1–20. https://doi.org/10.1007/978-981-10-5026-8_2

12. Ранівала, Хемант та ін. «Розробка економічно ефективних та масштабованих безсерверних застосунків». Міжнародна конференція IEEE з хмарної інженерії (IC2E) 2021 року, 2021, с. 95–102. https://doi.org/10.1109/IC2E52221.2021.00028.

13. Лю, Ян та ін. «Розуміння безсерверних обчислень: погляд з розподілених систем». ACM SIGOPS Operating Systems Review, т. 53, № 1, 2020, с. 35–41.

14. Ішакян, Ватче, Вінод Мутхусамі та Александер Сломінський. «Обслуговування моделей глибокого навчання на безсерверній платформі». Міжнародна конференція IEEE з хмарної інженерії (IC2E) 2018 р., 2018, с. 257–262. https://doi.org/10.1109/IC2E.2018.00039

15. Гулісано, Вінченцо та ін. «StreamCloud: еластична та масштабована система потокової передачі даних». IEEE Transactions on Parallel and Distributed Systems, т. 23, № 12, 2012, с. 2351–2365. https://doi.org/10.1109/TPDS.2012.21

16. Тірумуруганатан, Суділа Рой та ін. «Курування даних за допомогою глибокого навчання: опитування». Праці фонду VLDB, т. 14, № 12, 2021, с. 3190–3202. https://doi.org/10.14778/3476311.3476381

17. Анантхакрішна, Рамеш та ін. «Усунення нечітких дублікатів у сховищах даних». Матеріали 28-ї Міжнародної конференції з дуже великих баз даних (VLDB), 2002, с. 586–597. https://doi.org/10.14778/645806.669434.

**References**

1. Baldini, Ioana, et al. “Serverless computing: Current trends and open problems”. Research Advances in Cloud Computing (2017): 1–20. https://doi.org/10.1007/978-981-10-5026-8_1

2. Spillner, Josef, et al. “FaaSdom: A benchmark suite for serverless computing”. 2019 IEEE/ACM International Conference on Utility and Cloud Computing (UCC). IEEE, 2019. https://doi.org/10.1109/UCC48980.2019.00015

3. Jonas, Eric, et al. “Cloud programming simplified: A Berkeley view on serverless computing”. arXiv preprint arXiv:1902.03383 (2019). https://doi.org/10.48550/arXiv.1902.03383

4. Wang, Lei, et al. “Peeking behind the curtains of serverless platforms”. Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC 18). 2018. https://doi.org/10.5555/3291168.3291184

5. Reuter, Christian, et al. “A pattern-based approach to API error resilience for cloud-native systems”. Proceedings of the 2020 IEEE International Conference on Cloud Engineering (IC2E). IEEE, 2020. https://doi.org/10.1109/IC2E48795.2020.00011

6. Van Eyk, Eelco, et al. “The SPEC-RG Reference Architecture for FaaS: From Microservices and Containers to Serverless Platforms”. IEEE Internet Computing, vol. 23, no. 6, 2019, pp. 7–18. https://doi.org/10.1109/MIC.2019.2952061

7. Richardson, Chris. Microservices Patterns: With examples in Java. Manning Publications, 2018. https://doi.org/10.1007/978-1-4842-5574-3_3

8. Medel, Víctor, et al. “Middleware support for context-aware applications with reusable components”. Journal of Systems and Software 134 (2017): 181–197. https://doi.org/10.1016/j.jss.2017.08.009

9. Resende, Pedro, et al. “A fault-tolerant architecture for REST-based APIs”. Future Generation Computer Systems 108 (2020): 422–435. https://doi.org/10.1016/j.future.2020.03.005

10. Spillner, Jörg, and Mirko Westerheide “Serverless Data Processing with AWS Lambda and Apache Spark”. Journal of Cloud Computing, vol. 9, no. 1, 2020, pp. 1–14. https://doi.org/10.1186/s13677-020-00172-0

11. Baldini, Ioana, et al. “Serverless Computing: Current Trends and Open Problems”. Research Advances in Cloud Computing, edited by Rajkumar Buyya et al., Springer, 2017, pp. 1–20. https://doi.org/10.1007/978-981-10-5026-8_2

12. Raniwala, Hemant, et al. “Design of Cost-Efficient and Scalable Serverless Applications”. 2021 IEEE International Conference on Cloud Engineering (IC2E), 2021, pp. 95–102. https://doi.org/10.1109/IC2E52221.2021.00028

13. Liu, Yang, et al. “Understanding Serverless Computing: A View from Distributed Systems”. ACM SIGOPS Operating Systems Review, vol. 53, no. 1, 2020, pp. 35–41.

14. Ishakian, Vatche, Vinod Muthusamy, and Aleksander Slominski. “Serving Deep Learning Models in a Serverless Platform”. 2018 IEEE International Conference on Cloud Engineering (IC2E), 2018, pp. 257–262. https://doi.org/10.1109/IC2E.2018.00039

15. Gulisano, Vincenzo, et al. “StreamCloud: An elastic and scalable data streaming system”. IEEE Transactions on Parallel and Distributed Systems, vol. 23, no. 12, 2012, pp. 2351–2365. https://doi.org/10.1109/TPDS.2012.21

16. Thirumuruganathan, Sudeepa Roy, et al. “Data curation with Deep Learning: A Survey”. Proceedings of the VLDB Endowment, vol. 14, no. 12, 2021, pp. 3190–3202. https://doi.org/10.14778/3476311.3476381

17. Ananthakrishna, Ramesh, et al. “Eliminating fuzzy duplicates in data warehouses”. Proceedings of the 28th International Conference on Very Large Data Bases (VLDB), 2002, pp. 586–597. https://doi.org/10.14778/645806.669434