Rev. Roum. Sci. Techn.– Électrotechn. et Énerg.
Vol. **71**, *2*, pp. 293–298, Bucarest, 2026

ARCHITECTURE FOR INTEGRATING AIR QUALITY DATA FROM MULTIPLE SOURCES

MARIUS MARIAN DINU, ADRIANA OLTEANU, ANCA DANIELA IONIȚĂ

Keywords: Data integration; Air quality monitoring; Multi-source data; Data heterogeneity; Software development kit (SDK) architecture.

Air quality (AQ) monitoring has become an imperative component of public health strategy, yet traditional monitoring paradigms need improvement, and legacy systems designed for specific regional compliance struggle to keep up with the complexity of the problem. The study of existing air quality data reveals a variety of sources, data formats, and diverse parameter definitions across data providers. This paper proposes an architectural blueprint for a layered system capable of integrating air quality data, streamlining access to data from multiple sources into a single location, and reducing data heterogeneity to a common ground. The solution presented is implemented as an iOS software development kit (SDK) to demonstrate its practicality, with the long-term objective of transforming a disparate data landscape into a cohesive platform.

1. INTRODUCTION

Air pollution is one of the primary risk factors for human health, reason why it is fundamental that authorities use historical data to analyze pollutant trends, and determine public health risks, as well as the effectiveness of adopted environmental policies [1].

A considering flaw in the current monitoring systems is the disconnect between different data providers, the most noticeable being between the households and the enterprise entities (including the municipalities), inducing ambiguity in domestic decision activities [2]. So, a substantial barrier to an effective environment strategy is the governance over the decentralized management of the data, which causes fragmentation, forcing each entity to prioritize its own needs in the short term, but ignoring the holistic view [3].

While the proliferation of low-cost, real-time Internet of Things (IoT) sensors has addressed data scarcity, it has created new challenges of data heterogeneity and ambiguity, a dilemma between rich data input and poor insight output, suggesting that data is useless in isolation, without synthesis [4,5]. A consequential research gap exists for a unified structure capable of synthesizing data from multiple sources. Monitoring air quality process is not only about determining the current state but also about forecasting the next move based on significant factors [6,7].

In the long run, our research aims to perform air quality data fusion based on multiple open data sources, and it starts with the integration of air quality data originating from several open data providers. The definition of the term data fusion varies depending on the level of generality and is adapted to the specific needs of the research domain. In the general context of data analysis – regardless of the nature of the data – data fusion refers to a set of tools used for the joint analysis of data from multiple sources, whose interaction enables the generation of information that cannot be obtained from the individual sources alone [8]. According to [9], from a taxonomic perspective, data fusion can be classified into three levels of abstraction: the observational level (involves applying integration methods directly to raw or unprocessed datasets), the intermediate/feature level (operates on state vectors extracted from each dataset), and the high/information level (where the decisions resulting from the processing of each data block are fused). In this stage, we present an architecture for the observational level to integrate data obtained through various APIs.

The core technical contribution of the work lies in the establishment of a unified air quality data access layer that abstracts the integration efforts, ensuring data management through centralized storage, while optimizing user experience by autonomously handling location services. The modular design approach promotes flexibility and scalability, enabling seamless extension of data sources and functional capabilities without compromising architectural stability.

The paper is structured to first place the importance of air quality monitoring for public health and to expose the limitations of current systems, highlighting the problems related to the lack of interoperable data and decentralized custom systems that process them. Next, the paper presents a theoretical approach to the problem of data integration, followed by implementing these concepts into an architectural blueprint that can be followed and expanded, depending on the needs.

2. BACKGROUND AND KEY CONCEPTS

This section aims to establish theoretical concepts and lay down the necessary background for understanding the proposed architecture. The section begins by defining different perspectives on how the data can be integrated as one stream. Going further, the section examines the dominant issues of the current data sets and their causes.

2.1 INTEGRATION APPROACHES

In a general approach, data integration involves combining a set of selected information systems (often independent and heterogeneous) into a new, unified information system that provides an overview of the data and facilitates access to complementary information. An integrated data system thus creates the appearance of interacting with a single entity, offering a logically consistent and homogeneous view of the data, regardless of its underlying heterogeneous distribution.

To achieve this goal, data must be transformed using consistent abstraction principles, while also detecting and resolving conflicts related to structure and semantics. Typically, information systems are not originally designed for integration, which means they may require additional adaptation processes, varying on the architecture of the information system, the content and functionality of the components, and the types of data being exchanged (structured, semi-structured, or unstructured).

Elements that contribute to heterogeneity in unified data

Department of Automation and Industrial Informatics, Faculty of Automatic Control and Computer Science, National University of Science and Technology POLITEHNICA Bucharest, Romania.
E-mails: marius_marian.dinu@upb.ro, adriana.olteanu@upb.ro, anca.ionita@upb.ro

DOI : 10.59277/RRST-EE.2026.2.21

management include: the hardware units and operating systems managing the information, data models, validation schemas, and data semantics [10]. From an architectural perspective, we can outline an overview of the different approaches to data integration, with classification based on the level of abstraction at which integration occurs. Considering that information systems can be described using a layered architecture (Fig. 1), raw data resides at the lowest layer, where it is managed through storage systems.

Middleware acts as a bridge between the application and the database, handling data transactions, while applications operate at the upper layer, featuring user interfaces for external interaction.

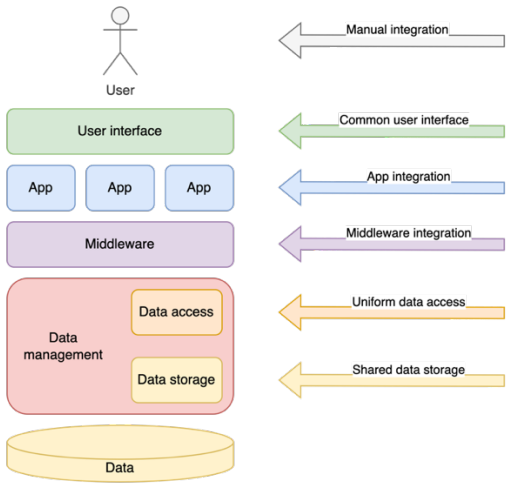

Fig. 1 – General approaches to data integration at different architectural levels.

2.2 DATA SOURCES

A typical data integration scenario involves multiple data sources that have been developed independently, often fragmented or distributed across autonomous systems, making data aggregation a challenging process. In addition to the decentralized nature of the sources, the integration process is further complicated by the fact that the data is organized within each system differently, each based on distinct conceptual formatting models.

Even though structural conflicts in data can be addressed using tools that convert data into a homogeneous format, the issue of semantic heterogeneity persists [11]. Semantic differences are inevitable and reflect the way data has been modeled according to the perspective of the developers who originally designed the dataset for a specific application. Additionally, stored data often contains redundancies that overcrowd storage resources without contributing real value to the final output.

Regardless of whether datasets appear similar in terms of syntax, their semantics must be carefully examined. Once the representation method for each property is identified, the normalization stage can begin. This process involves homogenizing the data by translating it into a common base, using a set of rules that preserve the meaning and integrity of the original values.

The collection and management of environmental data is essential for developing effective environmental protection strategies. In line with political and economic objectives, the need has emerged for information networks capable of managing environmental data (such as water, air, and soil quality) [12]. These systems typically process large volumes of data, aiming to simulate aspects of the natural environment. Later, the data is used for a variety of purposes, from informing the public to supporting decision-making systems, which means that databases must support complex queries with process scalability and storage capacities.

A wide range of environmental data is collected, generating valuable insights for decision-makers, citizens, and businesses. In most cases, data is collected in a distributed and heterogeneous manner, organized under different data models, and sourced from various platforms (both hardware and software), depending on the collecting entity. Additionally, the data tends to have a complex structure, reflecting the specific environmental element it describes.

The objects that model the acquired data exhibit spatial-temporal properties, being specific to a certain geographic location, with values that evolve over time. The data may be affected by noise or acquisition anomalies, and therefore, it must be automatically filtered to avoid impacting the results of subsequent processing. To maximize the utility of this data, the development of a universal system has been proposed – one that interconnects existing local databases and provides shared access to them.

The nature of the data varies depending on the phenomena being studied and their distribution in the environment. This type of data tends to acquire Big Data characteristics, incorporating various challenges in terms of management and processing [13]. Big Data refers to data that comes in high variety, large volumes, and at high velocity – specifically, large and complex datasets, often from new sources, that could not be handled using traditional methods.

Variety refers to the multiple types of data available for acquisition, a characteristic abundantly present in the environmental domain. Volume reflects the sheer amount of data, which can reach enormous levels. Velocity refers to the high speed at which environmental stimuli are captured, further burdening storage systems.

This has highlighted the need for a more systematic approach to managing data at this scale. Later, two additional characteristics were added to Big Data: value and veracity. Value refers to the intrinsic, hidden worth of data, which must be uncovered to be useful. Veracity concerns the level of trust or uncertainty associated with the data.

3. SDK FOR DATA INTEGRATION

The proposed solution involves the development of an SDK for selected Apple platforms (iOS, iPadOS) aimed at streamlining air quality data collections. This SDK integrates multiple public services (via Application Programming Interface – API) and consolidates them into data structures specific to the Swift programming language. The tool facilitates interactions with the mobile device's location module, as well as the accumulation and processing of data, allowing developers to focus primarily on implementing user-facing application functionalities.

The implementation of an integration solution can take various forms, with the approach presented in this study being influenced by the available technical expertise and the accessibility of relevant datasets. Consequently, while the proposed solution may appear restrictive, it provides valuable insights into addressing the challenges associated with environmental data integration.

3.1 TECHNOLOGIES

The implementation of the SDK is carried out using Apple's integrated development environment, Xcode. The

programming language used is Swift, while the demonstrative application is built with SwiftUI. Unlike application development, an SDK involves designing and implementing software components that can be integrated into multiple applications, regardless of the project's structure or domain-specific requirements.

More specifically, this process entails defining a framework project, which, upon compilation, generates a binary file. This file enables access to the public components within integrating applications. A framework is a hierarchical structure that encapsulates shared resources, such as libraries and files, within a single package. These resources can be simultaneously utilized by multiple applications, with the operating system managing their loading into memory as needed.

Frameworks serve a similar purpose to shared libraries – either static (embedded within the application) or dynamic (referenced externally), by providing a collection of routines that can be invoked by an application to perform specific tasks [14]. This bundling facilitates the installation and management of additional resources required during the development process. In essence, a framework exposes a set of specific functionalities that can be selectively accessed through its public interface.

Development in Xcode itself relies heavily on the use of frameworks, ranging from the foundational components of the operating system (iOS SDK) to development kits that provide specialized features. For instance, in the current approach, CoreLocation serves as the bridge between the device's location system and the developer, while BackgroundTask enables the scheduling of processes to run regardless of the application's state-whether in the foreground, background, or suspended.

3.2 SOFTWARE ARCHITECTURE

The proposed SDK, named MaqiSDK (Multiple Air Quality Indexes SDK), is an integrated solution designed for iOS mobile application developers. It consolidates multiple public APIs into a unified, language-specific interface (Swift), streamlining the integration process by abstracting API requests and reducing overall complexity.

In addition to facilitating easy access to environmental data, the SDK manages communication with the device's location services, enabling the provision of location-specific information. It also includes mechanisms for data storage and processing. The solution serves as a practical example of data fusion concepts and illustrates an approach to building an integrated system for environmental information and protection.

MaqiSDK does not include visual components typically used for graphical user interface (GUI) development in commercial applications. Instead, it is the responsibility of the integrator to design and implement appropriate data presentation for end users. However, the project includes a demonstrative application that integrates the SDK into an intuitive interface, showcasing the integration process and exposing the available capabilities of the SDK.

The architecture is modular (Fig. 2), composed of independent components with clearly defined responsibilities – such as interacting with specific APIs, managing location services, storing values, or performing comparisons. These modules interoperate to achieve the desired outcomes; for example, using the device's location to query air quality services.

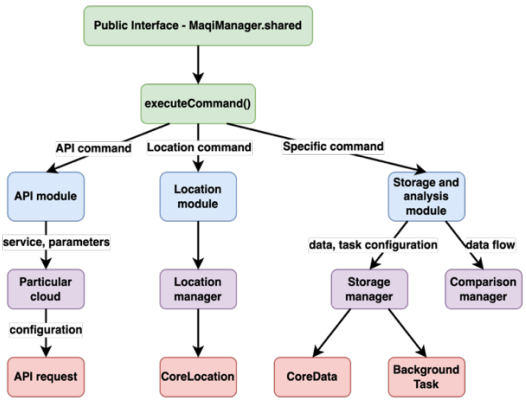

Fig. 2 – The SDK's layered architecture.

3.2.1 STAKEHOLDERS

The environmental data integration SDK will serve multiple interests across various stakeholders involved throughout its lifecycle – from concept to implementation and maintenance.

Initially, we can identify the end users of the product, who can be divided into several categories:

- application developers (the primary clients, who will integrate the SDK into their applications; they require comprehensive documentation, usage examples, and integration support).
- environmental researchers (indirect users, as they interact with applications built using the SDK for data collection, analysis, and visualization).
- consumers (also indirect users, who access air quality information through such applications).

Furthermore, in the event of open-source development and collaboration, the SDK could attract interest from the developer community, who may contribute code, report issues, and suggest improvements.

More broadly, we can also identify data providers, like organizations or institutions that supply environmental data streams, including specialized bodies such as environmental organizations, the Ministry of the Environment, or the European Environment Agency, which could offer valuable input for enhancing the SDK's data management capabilities.

3.2.2 SDK ARCHITECTURE

The architecture encapsulates all internal components involved in any form of data manipulation and exposes a single public class MaqiManager, which serves as the central coordinator for all interactions between the SDK and the developer. Furthermore, integrators have access to the data structures used within the SDK's commands, allowing for seamless integration and customization within their own applications.

MaqiManager provides a single shared instance through which all request flows are handled via the public method executeCommand(COMMAND). This universal method delegates the execution of the requested action to the appropriate module, as specified by the value of the COMMAND parameter. All available commands are grouped by module to ensure a clear separation of functionalities. The method is designed to execute one command per invocation but can be called repeatedly in sequence to perform multiple operations.

The action performed by this function is not executed instantaneously, and it requires the specification of a parameter to identify and resume the execution once a response is received. This parameter, referred to as a callback (a function passed as an argument to another function, which is then invoked within the outer function to complete a specific routine or action), is mandatory in all command invocations due to the asynchronous nature of the function. The callback is used to return the result of the command execution. The response is delivered in a structured format that indicates either success or failure, along with any relevant data (e.g., success: the actual location; error: permission not granted).

At a deeper level within the SDK, a set of protocols defines the required structure of each module in terms of the methods it must implement. The implementation of these lower-level modules is responsible for handling the execution of the respective commands. Each method within a module corresponds directly to an external command, ensuring a clear and consistent mapping between the command interface and its internal logic.

At the level of the modules associated with the APIs, a dedicated support class is defined for each, forming a pseudo-layer generically referred to as a service. This layer acts as a bridge between the public commands and their actual implementation. Its main responsibility is to invoke the methods defined in the lower-level cloud class and to process the returned data accordingly.

Within the cloud class (responsible for API request configuration), the requests to the API services are parameterized by specifying the target service and the parameters with which it will be called. This class returns the raw response from the API, including both the data and any associated error.

At the lowest level, there is a base class responsible for executing the actual API request. This class receives a customized request containing details of a specific API method, constructs the corresponding web request, and initiates the call to the public service. From a technical standpoint, the API request involves creating and configuring a URL session by specifying the protocol (HTTP/HTTPS), the target service, the accessed resource, and the method-specific parameters.

Each layer includes a set of validations and operations, specific conditions, along with error handling mechanisms for potential issues.

For instance, at the lowest level, the system checks the structural correctness of the request, verifies the reception of data in the response, and handles timeouts during response retrieval.

In the case of other available modules (such as location, storage, and analysis), manager classes are responsible for handling invoked commands and managing communication with the functional support components provided by the operating system, such as location services, the local database, and background processing tasks. The manager class associated with the comparison command is specifically tasked with processing input data received in various formats, extracting common properties, and returning the results in a structured format suitable for analysis.

To safeguard user privacy, the location services module operates on a permission-first basis, requesting access only when necessary. Geolocation data is processed ephemerally; the coordinates are used strictly to query the nearest air quality stations. Most of the APIs do require registration to access data, so the calls are linked to a certain API key. From a security perspective, this matter still needs investigation since, for now, the data is only used for internal testing.

3.2.3 DATA

To overcome the various formats, the SDK implements a unified data model, decoupling from the data provider's specifics, that is designed to support standard criteria for air quality data. For this purpose, the structure contains the following elements:

- API identifier (string).
- air quality index data value (integer).
- date of the measurement (date).
- dominant pollutant (string).
- air pollutants with their values (string: double dictionary).

Data ingestion is handled via provider-specific adapters that map proprietary fields to the defined standard model. Emphasizing, the mapping logic is custom, manually made for the current data sources, and it consists of aligning: measurement units, pollutant naming, and AQI scales.

The interaction flow with the SDK is handled by the public singleton available through the MaqiManager class. Singleton represents a software design pattern that restricts the instantiation of a class to a single instance, ensuring global and controlled access to that instance. The available commands follow a uniform structure and target specific actions within individual modules.

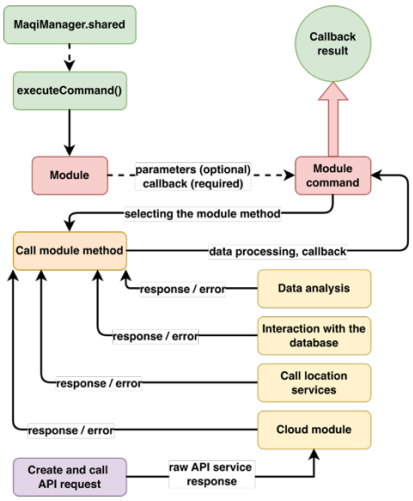

Fig. 3 – SDK data flow scheme.

Figure 3 illustrates the data flow through the SDK, from the invocation of a command associated with a module to the return of a response. Once the response is received, it becomes the integrator's responsibility to handle and process it accordingly.

The data exchanged between the SDK and the external environment consists of either generic data types (such as integers, strings, etc.) or custom structures defined within the SDK and exposed to the integrator (for example, the response containing the status of location services).

3.2.4 SDK INTEGRATION APP CAPABILITIES

The SDK is delivered as an *XCFramework*, an official Apple binary format, allowing it to work seamlessly across diverse development environments. This format provides native support for multiple architectures (physical devices

and simulators) and platforms (iOS, iPadOS) within a single bundle. The distribution pipeline leverages Swift Package Manager (SPM) to serve the tool as a binary target. This approach abstracts the underlying implementation details, protecting the proprietary code, while offering developers a familiar, dependency-managed integration experience.

The SDK supports iOS/iPadOS versions 15.0 and above, requires Xcode version 14.0 or later as the development environment, and uses Swift 5 or newer as the programming language. Integration involves embedding the SDK as a framework within the target application project.

This tool requires an active internet connection to function properly, as it interacts with a variety of services accessed via APIs.

Given the SDK's reliance on sensitive user data, the integrator must add the NSLocationWhenInUseUsageDescription key to the Info.plist file, with its value describing the reason for requesting location access, which will be displayed to safeguard the user during the authorization process.

Additionally, if location updates are required while the app is in a suspended state (i.e., the app is no longer in the foreground but has not been terminated), the developer must enable the Background Modes capability and check the Location updates option. Capabilities allow the integration of specific system-level features that support app functionalities. Enabling this setting allows the operating system to resume the app in the background and perform specific actions, even when it is not in the foreground.

3.2.5 COMMANDS AND RESPONSES

Access to the SDK methods is obtained through the manager class by calling the command execution method and specifying the desired service. This scenario is only possible after the SDK has been imported into the file (import MaqiSDK). For demonstration purposes, the following example illustrates a request for location permissions (Fig. 4).

```
MaqiManager.shared.executeCommand(
 .LOCATION_MODULE(//module selection
  .CMD_REQUEST_PERMISSIONS(completion)))
//method specification
//mandatory callback, some methods may also
have other specific parameters.
```

Fig. 4 – Location module command call, for requesting permissions.

To generalize, we can consider the following calling structure presented in Fig. 5.

```
.MODULE(.COMMAND(parameters, completion)).
```

Fig. 5 – Generic SDK module command call.

Regardless of the command, the response is returned using callback functions, which are defined as generic function types with a single parameter (the response) and no return value. These callbacks provide the response status (success or failure), and depending on the specific needs of a command, the success and error values are customized accordingly.

To complement the previous example, the code snippet in Fig. 6 highlights the response returned by the method that retrieves the authorization status of location permissions.

```
MaqiResponse<
     [(LocationPermissionType,
       LocationPermissionStatus)],
     MaqiResponseError<LMError>>)-> Void
```

Fig. 6 – Location permissions response model.

MaqiResponse represents the generic response type, customized so that on a successful command, it returns the status of the location services at both the device and application level, and in case of failure, it returns a module-specific error. In a generalized form, we can compact the response structure as in Fig. 7.

```
RESPONSE<
     [REQUESTED DATA],
     ERROR<MODULE SPECIFIC>>) -> Void
```

Fig. 7 – Generic SDK command response structure.

4. SDK TESTING FOR AIR QUALITY DATA

Testing the SDK involves verifying the specific behavior of each method within every module. Given the protocol-oriented structure, simulating a desired behavior is relatively straightforward.

4.1 UNIT TESTING

The unit tests for the SDK's commands are achieved by initializing the modules with a set of services that implement the required functions using predefined behavior.

For each cloud class in every module, a mock class was defined to simulate the actual behavior of the API, without requiring actual API calls (to avoid exceeding the available API quotas). To validate the correct interpretation of data, the test project includes JSON files containing valid API responses for both successful and error cases. The implementation of the cloud class methods retrieves the simulated response from these JSON files and passes it to the corresponding service class for processing, thereby following the complete data flow cycle within the SDK.

Fig. 8 schematically illustrates the logic of unit test implementation and the simulation of data.

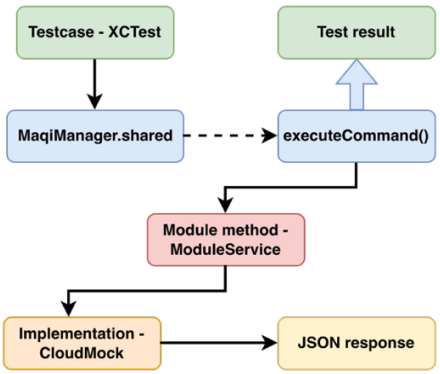

Fig. 8 – The logic of implementing a unit test.

In designing the tests, both success and failure cases were configured for each available command in the public interface, covering all the modules. These test cases aim to verify the proper handling of public commands and the correct return of data relevant to the requested query, while respecting the asynchronous nature of the methods. To summarize, in this phase, the API data parsing is verified, ensuring correct deserialization of heterogeneous JSON structures from different providers and the SDK's capacity to handle errors (HTTP errors, timeout scenarios, malformed payloads, quota limits exceeded).

4.2 PROOF OF CONCEPT

A Proof-of-Concept (PoC) application was implemented to test the API calls, that expands all the SDK's available air

quality data sources, handling their call parameters and responses. Each command can be configured and run manually, consuming data directly from the APIs. The raw data goes through the SDK logic, and it is returned in a normalized form, common to all the data sources.

In this context, the gathered data is focused on monitoring the air quality of a certain location by integrating the input from a range of data sources, resulting in a holistic view of the state of pollution and the specific pollutants present at that moment [15]. Fig. 9 lists the processed APIs, including the air quality indexes (AQI), along with the computed value of the air quality and the main air pollutant.

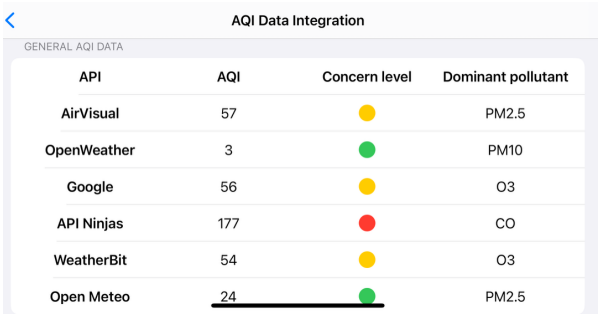

Fig. 9 – Air quality data visualization in PoC.

5. CONCLUSIONS

This paper provides a context of air quality monitoring issues, painting the current data landscape as a mosaic of isolated pieces, being part of the same view, but limiting its capabilities to custom scenarios. By providing an architectural structure that can be replicated into any number of programming languages, developers can move beyond the struggle to organize the data to build valuable tools.

This air quality data integration framework was designed to contribute to the core problem of data fragmentation, providing a scalable and developer-centric solution engineered to harmonize and synthesize data from multiple heterogeneous data sources. While notable challenges in data standardization remain, the SDK presented provides a blueprint for the field, marking a step away from isolated data points.

Looking ahead, this model opens new directions for exploration; future work could focus on developing autonomous processes that integrate air quality data from different providers and provide predictive insights on the subject [16].

Future research will focus on enhancing the SDK's autonomy and resilience, the primary objective being the implementation of an automatic parser capable of dynamic structural discovery. The validation strategy will also incorporate integration pipelines for automated regression testing under a wide range of conditions. Finally, to address the evolving privacy and security challenges, future versions will take into consideration privacy mechanisms dealing with geospatial queries and informing end users about all the parties involved in the data exchange.

CREDIT AUTHORSHIP CONTRIBUTION STATEMENT

MARIUS MARIAN DINU (corresponding author): conceptualization, methodology, writing original draft, software, investigation, writing submitted version, implementation, validation.

ADRIANA OLTEANU: manuscript structuring, investigation, methodology, supervising, review, and editing.

ANCA DANIELA IONITA: supervising, methodology, figure curation, manuscript structuring, writing, reviewing, and editing, checking references.

Received on 20 November 2025

REFERENCES

1. G. Năstase, A. Șerban, A.F. Năstase, G. Dragomir, A.I. Brezeanu, *Air quality, primary air pollutants and ambient concentrations inventory for Romania*, Atmospheric Environment, **184**, pp. 292–303 (2018).
2. O. Yildiz, H.S. Sucuoglu, *Development of real-time IoT-based air quality forecasting system using machine learning approach*, Sustainability, **17**, pp. 1–10 (2025).
3. R.N. Pietraru, A. Olteanu, I. Adochiei, F. Felix, *Reengineering indoor air quality monitoring systems to improve end-user experience*, Sensors, **24**, pp. 1–10 (2024).
4. T. Bidilă, R.N. Pietraru, A.D. Ioniță, A. Olteanu, *Monitor indoor air quality to assess the risk of COVID-19 transmission*, 2021 23rd International Conference on Control Systems and Computer Science (CSCS), Bucharest, Romania, pp. 1–6 (2021).
5. D. Carstoiu, E. Oltean, G. Gorghiu, A. Olteanu, A. Cernian, *Approaches in wind field modeling and air quality monitoring systems*, Bulletin UASVM Horticulture, **65**, *2*, pp. 549–554 (2008).
6. P. Wang, Q. Zhang, J. Dai, S. Wang, H.H. Cai, *Integration needs and challenges for green and smart transformation of port industry based on multi-source data*, Rev. Roum. Sci. Techn. – Électrotechn. et Énerg., **69**, *2*, pp. 1–10 (2024).
7. J.J. Hilly, K.R. Singh, P. Jagals, F.S. Mani, A. Turagabeci, M. Ashworth, M. Mataki, L. Morawska, L.D. Knibbs, R.M. Stuetz, A.P. Dansie, *Review of scientific research on air quality and environmental health risk and impact for PICTS*, Science of The Total Environment, **942**, pp. 1–10 (2024).
8. M. Cocchi, *Data fusion methodology and applications*, Data Handling in Science and Technology, Elsevier, **31**, pp. 1–300 (2019).
9. S.M. Azcarate, R. Ríos-Reina, J.M. Amigo, H.C. Goicoechea, *Data handling in data fusion: Methodologies and applications*, TrAC Trends in Analytical Chemistry, **143**, pp. 1–10 (2021).
10. P. Ziegler, K.R. Dittrich, *Data integration – problems, approaches, and perspectives*, Conceptual Modeling in Information Systems Engineering, Springer, pp. 39–58 (2007).
11. E.M. Zaman, S.M.K. Quadri, E.M.A. Butt, *Information integration for heterogeneous data sources*, IOSR Journal of Engineering, **2**, *4*, pp. 640–643 (2012).
12. C. Zhang, *Introduction to environmental data acquisition*, Fundamentals of Environmental Sampling and Analysis, John Wiley & Sons, Ltd, pp. 1–10 (2024).
13. P. Taylor, *Web APIs for environmental data – state of the art investigation*, Data61 CSIRO, pp. 1–10 (2016).
14. Apple Inc., *Framework programming guide*, pp. 1–100 (2025).
15. M.M. Dinu, A.M. Stanciu, A.D. Ioniță, *Software development kit for integration of environmental data*, 2025 15th International Conference on Advanced Computer Information Technologies (ACIT), Sibenik, Croatia, pp. 1–6 (2025).
16. K. Benmouiza, *Nonlinear clustered adaptive-network-based fuzzy inference system model for hourly solar irradiation estimation*, Rev. Roum. Sci. Techn. – Électrotechn. et Énerg., **68**, *1*, pp. 7–11 (2023).