K-HEALTH inAIR



K-HEALTHinAIR

Knowledge for improving indoor AIR quality and HEALTH

D4.9 K-HEALTHinAIR BigData platform (Version IV)

Project Title	Knowledge for improving indoor AIR quality and HEALTH
Project No	101057693
Contract start date	01/09/2022
Contract duration	48 Months

Document ID	K-HEALTHinAIR_D4.9 K-HEALTHinAIR BigData platform (Version IV)
Deliverable leader	Bull
Due date	31/08/2026
Deliverable date	24/08/2026
Dissemination level	PUBLIC

AUTHORS – CONTRIBUTORS

Name	Organization
Tomás Pariente, Miguel Ángel Mateo	Bull

PEER – REVIEWERS

Name	Organization
Simon de Leede	EMC
Jose Fermoso	CARTIF

DOCUMENT HISTORY

Version	Date	Author/Contributor (Organization)	Modifications	Status
V0.1	01.07.2026	Tomás Pariente / BULL	ToC	Draft
V0.2	23.07.2026	Miguel Angel Mateo / BULL	First Draft	Draft
V0.3	30.07.2026	Tomás Pariente, Miguel Angel Mateo /BULL	Final Draft for internal review	Draft
V0.3r	05.08.2026	Simon de Leede / EMC	Revision 1	Draft
V0.3r2	24.08.2026	Jose Fermoso/CARTIF	Review 2	Draft
V0.4	24.08.2026	Tomás Pariente / BULL	Final draft addressing internal review comments	Draft
V1	24.08.2026	Jose Fermoso/CARTIF	Final editing	Final version

Disclaimer

This deliverable may be subject to final acceptance by the European Commission. The information and views set out in this document are those of the authors and do not necessarily reflect the official opinion of the European Commission. Neither the Commission nor any person acting on the Commission's behalf may hold responsible for the use which may be made of the information contained therein.

Copyright message

Copyright message ©K-HEALTHinAIR Consortium, 2022-2026. This document contains original unpublished work or work to which the author/s holds all rights except where clearly indicated otherwise. Acknowledgement of previously published material and of the work of others has been made through appropriate citation, quotation or both.

TABLE OF CONTENT

1. Introduction 9
2. Summary of the Data Management Module status at M36 10
   2.1. D4.1 – Version I (Month 12) 10
   2.2. D4.7 – Version II (Month 24) 10
   2.3. D4.8 – Version III (Month 36) 11
3. Work carried out since M36 13
   3.1. Platform maintenance and data quality 13
   3.2. Platform handover 13
      3.2.1. Motivation and scope 13
      3.2.2. Architecture mapping 14
      3.2.3. AQDR docker compose deployment 15
      3.2.4. Data migration and restore procedure 16
      3.2.5. Handover outcome 16
   3.3. Deployment 17
      3.3.1. Prerequisites 17
      3.3.2. Directory Structure 17
      3.3.3. Services Deployed 18
      3.3.4. Configuration (.env file) 18
      3.3.5. First-Time Deployment 19
      3.3.6. Data Restore Procedure 20
      3.3.7. Verification and Post-Deployment Checks 21
      3.3.8. Notes and Limitations 21
      3.3.9. Auxiliary Scripts 21
4. Conclusions 23

Appendix I: CSV to InfluxDB Historical Data Loader Script 24

Appendix II: Mock Sensor APIs Server 31

LIST OF FIGURES

Figure 1: K-HEALTHinAIR AQDR docker-compose deployment directory structure. 18
Figure 2: K-HEALTHinAIR AQDR docker-compose building. 19
Figure 3: K-HEALTHinAIR AQDR docker-compose start. 19
Figure 4: K-HEALTHinAIR AQDR data restoring. 20
Figure 5: K-HEALTHinAIR CSV data loader examples of use data restoring. 24
Figure 6: K-HEALTHinAIR CSV data loader script. 30
Figure 7: K-HEALTHinAIR sensor server mock structure. 31
Figure 8: K-HEALTHinAIR sensor server mock example of use. 32
Figure 9: K-HEALTHinAIR sensor server mock example of use II. 32
Figure 10: K-HEALTHinAIR sensor server mock script. 38

LIST OF TABLES

Table 1: AQDR k8 Vs docker mapping. 14
Table 2: AQDR docker exposed services 15
Table 3: AQDR services deployed with docker-compose 18
Table 4: AQDR restoring data files 20
Table 5: Sensor server mock end points 31
Table 6: Mock connection parameters 32

ABBREVIATIONS AND ACRONYMS

ACRONYMS	DESCRIPTION
AQ, IAQ, OAQ	Air Quality, Indoor AQ, Outdoor AQ
AQDR	Air Quality Data Repository
ATC	Asthma Control Test
CAT	COPD Assessment Test
CAT	COPD Assessment Test
CSV	Comma-separated values
FOT	Forced Oscillation Technique
FS	Forced Spirometry
FS/FOT	Forced Spirometry/Forced Oscillation Technique
GMH	Global Mental Health
GFH	Global Physical Health
ICHOM	International Consortium for Health Outcomes Measurement
IPCHEM	Information Platform for Chemical Monitoring
LOINC	Logical Observation Identifiers Names and Codes
MiniAQLQ	Mini Asthma Quality of Life Questionnaire
mMRC	modified Medical Research Council (Dyspnea scale)
NIH	(The) National Institutes of Health
OCDE	Organisation for Economic Co-operation and Development
PaRIS	Patient-Reported Indicator Surveys
PREMs	Patient Reported Experience Measures
PREMs	Patient-Reported Experience Measures
PROMIS	Patient-Reported Outcomes Measurement Information System
PROMs	Patient Reported Outcome Measures
REDCap	Research Electronic Data Capture
REST-API/RESTful	Representational State Transfer Application Programming Interface
SACQ	Self-Administered Comorbidity Questionnaire
SNOMED CT	Systematized Nomenclature of Medicine - Clinical Terms
SNOT	Sino-Nasal Outcome Test
TAI	Test of Adherence to Inhalers
VAS	Visual Analogue Scale
WHO-5	(The) World Health Organization-Five Well-Being Index
WHODAS	(The) World Health Organization Disability Assessment Schedule

EXECUTIVE SUMMARY

This deliverable D4.9: '*K-HEALTHinAIR BigData Platform (Version IV)*' reports on the final state of the Data Management Module component of the Platform for the K-HEALTHinAIR project at month 48. This complements the previous documentation delivered under Task 4.1: Platform specification and construction (M1-M48) (D4.1, D4.7 and D4.8).

Previous versions of this deliverable focused on the functional development of the platform's submodules, this final version concentrates on ensuring the continuity and sustainability of the platform beyond the K-HEALTHinAIR project. The main contribution of this task is the handover of the Air Quality Data Repository (AQDR) submodule to the project coordinator (CARTIF). This includes the transfer of the source code, publicly available at https://github.com/Bull-RD-Spain/khealthinair-aqdr, as well as a self-contained deployment mechanism based on Docker Compose, provided within the same repository. This work has been designed as an alternative to the Kubernetes-based production infrastructure managed internally by ATOS/BULL and documented in previous versions of this deliverable's series. This mechanism allows the AQDR to be deployed and operated on a single Linux virtual machine, without requiring access to, or knowledge of, container orchestration concepts, thus making the sustainability of the AQDR simpler and more cost-effective.

As part of this handover, a dedicated restore procedure was developed and executed to migrate the existing, populated production instance (the sensor registry and the complete historical air quality time series) into the coordinator's new environment, ensuring continuity of the platform's operational data rather than a clean-slate installation.

Two auxiliary scripts developed to support the deployment and testing of the AQDR are also documented: a CSV-to-InfluxDB historical data loader, and a mock server simulating three external sensor provider APIs (Inbiot, Qlair, Kunak), enabling local development and testing independently of the real external services.

The Discrete Air Quality Sampling Repository (DAQSR) and the Health Data Hub (HDH) remain outside the scope of this simplified deployment package. They continue to be deployable through the original Kubernetes-based descriptors documented in D4.7 and D4.8. Although they have also been extended to docker compose mechanism, it has not been properly tested beyond the start procedure and therefore it remains as a future, post-project activities in case it would be required.

1. Introduction

This document is the last of the series of four deliverables with the name "K-HEALTHinAIR Big Data platform" and continues to report on the implementation status of the *Data Management Module*, within the complete K-HEALTHinAIR Platform developed for the project.

The document reports on the work carried out in the last months of the project to deliver the final version of the Data Management Module, as well as the steps taken towards the sustainability of the repository through the final website hosted by the project coordinator, CARTIF:

- Chapter 1 introduces the deliverable objectives.
- Chapter 2 provides an overview summary of the work carried out until M36, as a baseline to understand the final work in the last year of the project.
- Chapter 0 reports on the steps carried out to the handover of the repositories to CARTIF.
- Chapter 3.3 describes the deployment of the ADQR module.
- Finally, Chapter 0 concludes the document with final comments on the work carried out.

2. Summary of the Data Management Module status at M36

Continuing from the previous report on the implementation of the Data Management Platform, the current document is the last of the deliverable series. This section summarises the work reported in the three previous deliverables belonging to the Data Management Module (D4.1, D4.7 and D4.8), which together documented the progressive design, implementation, and consolidation of the K-HEALTHinAIR BigData Platform over the first 36 months of the project, while sections 0 and 3.3 will explain the work carried out since then until the end of the project.

2.1. D4.1 – Version I (Month 12)

The first version of the deliverable, submitted in August 2023, established the foundational architecture of the platform following a microservices-based paradigm, with all components containerised using Docker and orchestrated through Kubernetes. At this early stage, three main components were defined:

- The Air Quality Data Repository (AQDR), composed of a Time-Series Database (based on InfluxDB), an Ingestor responsible for scheduled ETL pipelines extracting data from sensor APIs, and an API to enable data retrieval. This was, from the outset, the most mature component, already ingesting real indoor and outdoor air quality (IAQ/OAQ) data from several pilot scenarios.
- The Discrete Air Quality Sampling Repository (DAQSR), based on MinIO object storage, designed to host discrete sampling data (VOC, PM, microbiome, formaldehyde, PAH, radon, etc.) organised in a bucket-per-pilot, folder-per-scenario structure.
- The Health Data Hub (HDH), conceived from the start as the future repository for clinical data, based on HL7 FHIR standards (via HAPI-FHIR) and a dedicated API, but at this stage still undefined pending agreement with clinical partners on the data to be collected.

This first version also introduced two structural elements that would remain foundational throughout the project: the K-HEALTHinAIR ID Convention, a hierarchical coding system (pilot, scenario, sub-scenario, area, participant, and discrete-sampling identifiers) used to uniquely and consistently locate every sensor, sample, and participant across the six project pilots (AT, DE, ES, NL, NO, PL); and the Air Quality Data Model (AQDM), a harmonised naming and unit convention agreed between sensor providers to unify the parameters stored in the AQDR.

2.2. D4.7 – Version II (Month 24)

The second version, submitted in August 2024, marked a naming and conceptual shift: the platform component described in this deliverable series was renamed from "Open Access Platform" to the more precise Data Management Module, reflecting the fact that access to the data was still being controlled and restricted, pending future decisions on what would ultimately be made openly accessible.

Substantial technical progress was reported for the AQDR, which was expanded with two new sub-components:

- AQDR-Platform-DB, a new relational database (PostgreSQL) dedicated to maintaining the registry of installed sensors, delegating their maintenance directly to pilot-responsible

partners through CRUD operations exposed via the AQDR-API. This significantly reduced manual administration overhead and human error.
- AQDR-UI, a customised Grafana-based graphical interface replacing the previously used native InfluxDB dashboard, offering both general (cross-pilot) and pilot-specific dashboards for exploratory data analysis, monitoring, and visual comparison of pollutants.

The AQDR-Ingestor was also enhanced with a new data source (Kunak, an additional outdoor sensor deployed in the Norwegian pilot), an improved backfilling mechanism (the evolution of the earlier "back-up" strategy), a caching system for outdoor air quality data, and a documented set of unit-conversion factors to accommodate future sensor providers using different measurement units.

The DAQSR was reinforced with access control and security policies, assigning dedicated users and bucket-level read/write permissions to each sampling partner (WUT, CIEMAT, CSIC, NIO, MUW), ensuring that each institution could only access data relevant to its own pilot.

The HDH and a prospective Machine Learning (ML) Module remained undeveloped at this stage, with only preliminary discussions on clinical data modelling (questionnaires, wearables, clinical records) having taken place — this work was explicitly deferred to Version III.

Finally, D4.7 introduced a dedicated "Platform development" chapter, documenting for the first time the technologies, development methodology (devcontainers, SonarQube static analysis), CI/CD pipelines (GitHub Actions), and full configuration reference (environment variables, container ports, dependency versions) underpinning the platform — a structure that would be maintained and extended in subsequent versions.

2.3. D4.8 – Version III (Month 36)

By this stage, the AQDR had reached functional stability, requiring only minor corrections and usability adjustments based on partner feedback (e.g., warnings for data-point display limits, refined cascading filters). The DAQSR likewise remained largely unchanged, functioning as a stable archival repository, with the addition of a formal backup mechanism.

The central achievement of D4.8 was the effective implementation of the Health Data Hub (HDH), the platform's most complex and long-delayed component, whose development had been deliberately postponed until sufficient clinical data collection had matured across the pilots. The HDH was completed as two sub-modules:

- The Clinical Data Repository (CDR), based on HAPI-FHIR (a Java reference implementation of the HL7 FHIR standard), providing standardised, semantically interoperable storage for clinical data.
- The Clinical Data Repository API (CDR-API), a Java/SpringBoot RESTful wrapper around the CDR, implementing the transformation and validation logic needed to import data and export normalised datasets for analysis, grouped into Admin, Import, and Export endpoint categories.

A major part of D4.8 is dedicated to describing the data pipeline from REDCap to the CDR. Most pilots (with the exception of the Dutch pilot, for privacy reasons) used REDCap to collect clinical and health-related questionnaire data from enrolled participants. A standardised tabulated export format ("Data Export - KHIA - EVIDEN") was agreed to transfer this data into the CDR-API,

which performs validation, standardisation, and mapping into HL7 FHIR resources (Patient, Observation, Questionnaire, QuestionnaireResponse). The deliverable documents in detail the full set of clinical questionnaires used across pilots together with their mapping to standardised clinical terminologies (SNOMED CT, LOINC) wherever available. The questionnaires were organised into six parts: 1) Socio-demographic data; 2) Time spent in scenario; 3) PROMs; 4) PREMs; 5) Quality of Life; and 6) Mental Health. They include internationally recognised instruments such as SACQ, PROMIS Global-10, WHODAS-12, WHO-5, CAT, mMRC, TAI-12, MiniAQLQ, ATC, SNOT-22, EQ-5D-5L, and PHQ-9/PHQ-8.

D4.8 also introduced a platform-wide backup strategy based on Kubernetes Volume Snapshots, applying daily and monthly snapshot policies to the persistent volumes of both the AQDR and DAQSR components, addressing a previously unresolved data-resilience gap.

3. Work carried out since M36

During the last year of the project, the work related to the repositories was focused on improving the quality of the data collected, solving some issues regarding data collection (e.g., missing collection related to sensor failures), and most importantly, the handover of the platform to the Project Coordinator to ensure its sustainability.

3.1. Platform maintenance and data quality

The platform remained stable during the last year of the project. The main work carried out related to the platform maintenance consisted of giving support to pilots in relation to the data gathering, solving minor issues and improving performance. A summary of the main activities carried out is as follows:

Recovering missing data from sensors:

- The insertion date of a group of sensors in the NO pilot caused the ingestion process to fail. Once the error was identified and corrected, a significant volume of data remained un-ingested. Due to the limits imposed on the number of consecutive requests that the system can make to sensor APIs, this also caused the backfilling process to fail.
- To analyse the issue affecting the backfilling process, a mock sensor APIs (see Appendix II: Mock Sensor APIs Server) was implemented and used to identify the causes of the issue.
- Once the cause was identified, as removing the limit on consecutive requests was not feasible, it was decided to develop a new script for loading historical data from CSV files. This script resolved the issue and additionally enabled the platform to be populated with data originating outside the project and its sensor providers, thereby enriching the studies conducted.

AQDR API queries timeout:

- Queries to the InfluxDB database were improved to prevent timeouts in the AQDR API module when the data returned by SQL queries was excessively large.
- Several combinations of SQL query filtering and programmatic filtering of the retrieved data were tested.
- The performance of these queries has been improved; however, it remains the user's responsibility to limit, to some extent, the volume of data retrieved. The most effective way to achieve this is by also filtering queries by date.

3.2. Platform handover

3.2.1. Motivation and scope

The main work carried out was related to ensure the continuity and sustainability of the K-HEALTHinAIR BigData Platform beyond the project's active development period. This handover process, internally referred to as the KHIA Deployment initiative, addressed two complementary aspects:

- Opening the source code and making it available through a public code repository (https://github.com/Bull-RD-Spain/khealthinair-aqdr).
- Simplifying the deployment of the platform for portability to a single server or virtual machine with fewer resources than required in a Kubernetes environment. This simplification allows the handover of the platform to the project coordinator or any interested party to access, inspect, and redeploy the platform components independently of Kubernetes infrastructure.

Since the production environment of the K-HEALTHinAIR Platform had relied on a Kubernetes cluster managed (as described in the "Platform development" chapters of D4.7 and D4.8), an alternative, self-contained deployment mechanism based on Docker Compose has been developed and validated. This allows the platform to be deployed on a single Linux virtual machine, without requiring access to, or knowledge of, a Kubernetes cluster, container orchestration concepts, or the specific infrastructure tooling (Traefik ingress, cert-manager, Nexus registry integration, etc.).

This adapted deployment mechanism covers the Air Quality Data Repository (AQDR) submodule (https://github.com/Bull-RD-Spain/khealthinair-aqdr/tree/main/deployment/linux_server_restore), comprising its five constituent services: AQDR-Platform-DB, AQDR-TS-DB, AQDR-API, AQDR-Ingestor, and AQDR-UI. The Discrete Air Quality Sampling Repository (DAQSR) and the Health Data Hub (HDH) were not included in this simplified deployment package, although they have been also prepared for docker deployment as isolated services.

3.2.2. Architecture mapping

The Docker Compose deployment preserves the same microservices decomposition, container images, and inter-service dependencies already established in the Kubernetes-based production environment, while replacing the orchestration-specific constructs with their Docker Compose equivalents. Table 1 summarises this mapping.

Table 1: AQDR k8 Vs docker mapping.

Kubernetes construct	Docker Compose equivalent	Purpose
ConfigMap / Secret	.env file + environment	Externalised configuration (URLs, credentials, feature flags)
PersistentVolumeClaim (PVC)	Named Docker volume	Durable storage for databases, logs, and dashboards
Deployment / StatefulSet	service definition	Container lifecycle management
initContainer ("wait-for" pattern)	depends_on with condition: service_healthy	Ensures dependent services (e.g., AQDR-API) start only once their databases are ready

Kubernetes construct	Docker Compose equivalent	Purpose
livenessProbe / readinessProbe	healthcheck	Equivalent health verification logic (HTTP checks for API/UI, pg_isready for PostgreSQL, influx ping for InfluxDB)
Service (ClusterIP)	Docker internal network + service name resolution	Inter-container communication (khealthinair-net bridge network)
Ingress (Traefik + cert-manager, TLS, custom domains)	Direct host port publishing	Simplified access without reverse proxy or automated TLS certificate management
Container registry pull (newregistry.evidenresearch.eu)	${REGISTRY_HOST} variable	Same private image registry, parameterised for portability

This mapping reproduces the functional behaviour of the platform faithfully; same container images, environment variables, initialisation scripts, and health-check logic. While removing the operational dependency on a Kubernetes cluster and its associated ingress/TLS infrastructure. The result is a self-contained *docker-compose.yaml* file, together with *a .env* configuration template, and a set of initialisation scripts to reproduce, on first startup, the same database schema and bucket configuration already documented in D4.1, D4.7, and D4.8 (creation of the sensors PostgreSQL database and table, and of the *AIR_QUALITY* and *HEALTH_MONITORING* InfluxDB buckets).

3.2.3. AQDR docker compose deployment

The five AQDR services are deployed as follows:

Table 2: AQDR docker exposed services

Service	Image	Exposed port	Role
qdr-platform-db	postgres:16.4	5432	Sensor registry (relational DB)
aqdr-platform-db-ui	adminer:4.8.1-en_v1	8084	Administration UI for the sensor registry
aqdr-ts-db	influxdb:2.7.8	8086	Time-series storage for continuous IAQ/OAQ monitoring
aqdr-api	aqdr-api:latest	8083	Sensor CRUD operations and bulk data retrieval
aqdr-ui	aqdr-ui:latest	3000	Grafana-based dashboards

The *aqdr-ingestor* service, responsible for the scheduled ETL pipelines, was also parameterised in the *.env* template (periodicities, third-party API endpoints, InBiot/Qlair/Kunak credentials) and included in the compose stack, following the same configuration already described in D4.7.

All sensitive values (database passwords, InfluxDB admin token, API authentication secrets, SMTP credentials) were externalised into the .env file, which is excluded from version control and was provisioned individually for the coordinator's deployment target, mirroring the separation between source code and secrets already enforced in the Kubernetes Secret objects.

3.2.4. Data migration and restore procedure

To carry out the handover of a live, populated instance (rather than an empty deployment), a dedicated restore script (restore_aqdr.sh) was developed and executed to migrate the existing production data (sensor registry and historical air quality time series), into the new Docker Compose–based environment. The procedure performed the following steps:

1. Stopped and removed any previously running AQDR containers on the target host.
2. Updated the InfluxDB administration token in the local *.env* file to match the one associated with the source backup.
3. Started the two database services (*aqdr-platform-db, aqdr-ts-db*) and waited until both reported a healthy status.
4. Restored the PostgreSQL sensors table from a *pg_dump* export of the source database.
5. Restored the InfluxDB buckets (*AIR_QUALITY* and *HEALTH_MONITORING*) from a full influx backup/influx restore export, replacing the previous bucket content.
6. Started the remaining services (*aqdr-api*, *aqdr-platform-db-ui, aqdr-ui*).
7. Verified the final state of the stack and confirmed the resulting access URLs.

This restore mechanism complements the Kubernetes Volume Snapshot backup strategy already described in D4.8 (Section 4: Backups via Kubernetes Volume Snapshots). While the snapshot mechanism protects the production environment against data loss, the export/restore procedure described here provided a storage-backend-agnostic data package (a plain SQL dump plus an InfluxDB backup directory) that allowed the platform's data to be transferred to an entirely different infrastructure, namely a standalone virtual machine outside ATOS/BULL's Kubernetes cluster.

3.2.5. Handover outcome

The handover of the AQDR submodule to the project coordinator has been successfully completed:

The Docker Compose deployment package for AQDR (compose file, *.env* template, initialisation scripts, and restore script) have been prepared, tested, and validated on a Linux virtual machine, successfully reproducing the full AQDR functionality (sensor registry, continuous monitoring data ingestion, API access, and Grafana dashboards) outside the original Kubernetes environment.

The Docker images of the AQDR components have been transferred to the project coordinator through a private channel, ensuring the coordinator can pull and run the exact same validated

artifacts used in the ATOS/BULL managed production environment, without needing to rebuild them from source.

The source code was published in a public code repository, allowing the project coordinator and any third party to access, audit, and, if needed, rebuild the platform components independently.

The data migration and restore procedure described above was executed against the coordinator's target environment, successfully transferring the sensor registry and the historical continuous-monitoring time series into the new deployment.

3.3. Deployment

This chapter serves as a practical guide for deploying, operating, and restoring the K-HEALTHinAIR Air Quality Data Repository (AQDR) using the Docker Compose–based deployment package. Unlike the Kubernetes-based production deployment described in D4.7 and D4.8, this mechanism is designed to be operated on a single Linux virtual machine, without requiring familiarity with container orchestration concepts, and is intended for platform administrators taking over the operation of the AQDR submodule.

3.3.1. Prerequisites

Before deploying the platform, the target host must have the following software installed:

- Docker Engine and the Docker Compose plugin.
- A local clone of the public source code repository, containing the source of the *aqdr-api* and *aqdr-ui* components, from which the corresponding container images must be built before deployment.
- Sufficient open ports on the host for the services that need to be reached externally (by default: 3000 for the dashboards, 8083 for the API, 8084 for the sensor registry administration UI, 8086 for the time-series database, and 5432 for direct PostgreSQL access if required).

3.3.2. Directory Structure

The deployment package must be organised on the target server following a fixed directory layout, expected both by *docker-compose.yaml* and by the *restore* script:

```
~/khia/
├── docker-compose.yaml
├── .env                                ← configuration file (see below)
├── restore_aqdr.sh                     ← data restore script
├── init-scripts/
│   ├── 00-aqdr-platform-db-init.sh     ← creates the 'sensors' DB user
│   ├── 01-aqdr-platform-db-init.sql    ← creates schema/table for sensor registry
│   └── aqdr-ts-db-init.sh              ← creates additional InfluxDB buckets
└── db_backups/
    ├── postgres/
    │   └── postgres_sensors_<DATE>.sql
    └── influx/
        └── backup_<DATE>/              ← InfluxDB native backup folder
```

Figure 1: K-HEALTHinAIR AQDR docker-compose deployment directory structure.

The *init-scripts/* files are mounted as read-only volumes into the *aqdr-platform-db* and *aqdr-ts-db* containers and are executed automatically, once, the first time each database container is initialised with an empty data volume — reproducing the same initialisation logic already used in the Kubernetes ConfigMaps described in D4.7 (Appendix II: Platform DB initialization).

3.3.3. Services Deployed

The *docker-compose.yaml* file defines six services, all connected through a shared internal bridge network (*khealthinair-net*):

Table 3: AQDR services deployed with docker-compose

Service	Purpose	Depends on
aqdr-platform-db	PostgreSQL sensor registry	
aqdr-platform-db-ui	Adminer UI for the sensor registry	aqdr-platform-db (healthy)
aqdr-ts-dbInfluxDB	time-series database	
aqdr-api	Sensor management & data retrieval API	aqdr-platform-db, aqdr-ts-db (healthy)
aqdr-ui	Grafana-based	dashboardsaqdr-ts-db (healthy)
aqdr-ingestor	Scheduled ETL pipelines	aqdr-platform-db, aqdr-ts-db (healthy)

Each service that depends on a database only starts once that database reports a healthy status, replicating the "*wait-for*" *initContainer* logic used in the Kubernetes deployment through the native Docker Compose depends_on *condition: service_healthy* mechanism. Health status is determined through the same checks used in production: *pg_isready* for PostgreSQL, *influx ping* for InfluxDB, and HTTP checks against each application's health endpoint for the API and UI.

Data persistence is guaranteed through four named Docker volumes (*aqdr-platform-db-data, aqdr-ts-db-data, aqdr-api-logs, aqdr-ui-data*), equivalent to the *PersistentVolumeClaims* used in the Kubernetes deployment.

3.3.4. Configuration (.env file)

All environment-specific and sensitive configuration is externalised into a single *.env* file, which must be created and edited before the first deployment and is not included in the public source code repository. It is organised into the following blocks, mirroring the configuration tables already documented in D4.7 (Table 5: Configuration variables AQDR):

- *aqdr-platform-db*: database name, admin user/password, sensor registry user password, and Adminer connection settings.
- *aqdr-ts-db*: InfluxDB initialisation mode, admin username/password, organisation name, default bucket, admin token, and any additional buckets to be created (e.g., *HEALTH_MONITORING*, reserved for future HDH integration).
- *aqdr-api*: log level, InfluxDB connection URL and measurement name, API context path and authentication secret, sensor registry connection details, and the base URLs of the third-party sensor provider APIs (InBiot, Qlair, Kunak).
- *aqdr-ingestor*: request limits, sensor device identifiers, and ETL pipeline periodicities (InBiot, Qlair, and outdoor data, in minutes) — configurable independently, as already described in D4.7.
- *aqdr-ui*: Grafana admin credentials, public-facing URL, data source connection details and authentication token, and SMTP configuration for notifications (optional, disabled by default).
- Image source: the *aqdr-api* and *aqdr-ui* images are not pulled from a private registry but built locally from the source code repository, using the Dockerfile provided for each component. The REGISTRY_HOST variable used elsewhere in this document is therefore replaced by locally tagged image names (e.g., *aqdr-api:local*, *aqdr-ui:local*), referenced accordingly in *docker-compose.yaml*.

Administrators must replace all placeholder values (passwords, tokens, API keys) with environment-specific secrets before deployment. It is strongly recommended to generate a new, unique InfluxDB administration token and API authentication secret for each independent deployment, rather than reusing values from the original ATOS/BULL-managed environment.

3.3.5. First-Time Deployment

To deploy the AQDR module from scratch (empty databases), the following steps are performed from within the ~/khia/ directory:

1. Clone the public source code repository and build the *aqdr-api* and *aqdr-ui* container images locally:

```
git clone <public-repository-url>
cd <repository-folder>/aqdr-api && docker build -t aqdr-api:local .
cd ../aqdr-ui && docker build -t aqdr-ui:local .
```

Figure 2: K-HEALTHinAIR AQDR docker-compose building.

2. Populate the *.env* file with the target environment's configuration values.
3. Start the full stack:

```
docker compose up -d
```

Figure 3: K-HEALTHinAIR AQDR docker-compose start.

4. Verify that all services report a healthy status: Access the platform through the exposed ports (Grafana dashboards on port 3000, sensor registry administration on port 8084, API documentation on port 8083).

On first startup, the initialisation scripts automatically create the sensors PostgreSQL database, its schema, and the InfluxDB buckets, reproducing an empty but fully functional AQDR instance.

3.3.6. Data Restore Procedure

When the goal is not an empty deployment but the migration of an existing, populated AQDR instance, the restore_aqdr.sh script automates the transfer of both databases from a set of backup files into the target Docker Compose environment. This is the mechanism used to carry out the actual handover described earlier in this deliverable.

- Required inputs:

Table 4: AQDR restoring data files

Argument	Description
<pg_backup.sql>	PostgreSQL dump of the sensor registry (pg_dump)
<influx_backup_dir>	Directory containing a full InfluxDB backup (influx backup)
< influx_token>	InfluxDB administration token associated with the source backup

- Usage:

```
bash ~/khia/restore_aqdr.sh \
     ~/khia/db_backups/postgres/postgres_sensors_<DATE>.sql \
     ~/khia/db_backups/influx/backup_<DATE> <INFLUX_TOKEN>
```

Figure 4: K-HEALTHinAIR AQDR data restoring.

- Execution steps performed by the script:
  1. Validation: checks that all required arguments and files exist, and that *docker-compose.yaml* and *.env* are present in the expected location.
  2. Container cleanup — stops and removes any previously running AQDR containers, ensuring a clean restore.
  3. Token synchronisation: updates the InfluxDB admin token references in the local .env file to match the one associated with the provided backup, so the restored data can be authenticated correctly.
  4. Database bootstrap: starts only the two database services (*aqdr-platform-db, aqdr-ts-db*) and polls their health status for up to 200 seconds before proceeding.
  5. PostgreSQL restore: clears the existing sensors table content and reloads it from the provided SQL dump, reporting the number of restored sensor records.
  6. InfluxDB restore: copies the backup directory into the running container, removes any pre-existing *AIR_QUALITY* and *HEALTH_MONITORING* buckets, and performs a full

restore (influx restore --full), which recreates the buckets together with their historical data and metadata.

7. Application services startup — starts the remaining services (*aqdr-api, aqdr-platform-db-ui, aqdr-ui*).
8. Final verification: waits 30 seconds, prints the status of all containers, and reports the number of restored sensors together with the access URLs for Grafana, Adminer, and InfluxDB on the target host.

The script includes basic safeguards (parameter validation, health-check polling with timeouts, and explicit error messages) to reduce the risk of a partial or inconsistent restore, and can be safely re-run if an earlier attempt is interrupted, since it always starts by removing any previously running containers.

3.3.7. Verification and Post-Deployment Checks

After either a first-time deployment or a data restore, administrators should verify:

- That all six services report a *healthy/running* status (docker compose ps).
- That the sensor registry (accessible through Adminer on port 8084) reflects the expected number of registered devices.
- That the Grafana dashboards (port 3000) correctly display historical and, if the ingestor is enabled, newly incoming air quality data.
- That the API (port 8083) responds correctly to authenticated requests and that its Swagger UI documentation loads as expected.

3.3.8. Notes and Limitations

This deployment mechanism currently covers AQDR only; the DAQSR and HDH submodules remain deployable exclusively through the original Kubernetes descriptors documented in D4.7 and D4.8. This has been agreed within the project.

Since *aqdr-api* and *aqdr-ui* images are built locally from source rather than pulled from a private registry, administrators are responsible for rebuilding them whenever the source code repository is updated, in order to keep the deployed platform aligned with the latest published version.

The Docker Compose deployment does not include the TLS termination, custom domain routing, or automated certificate renewal features provided by the Traefik ingress and cert-manager setup used in the ATOS/BULL managed Kubernetes cluster. If public HTTPS access is required, a reverse proxy must be configured separately by the host administrator.

Secrets are managed exclusively through the *.env* file. Administrators are responsible for its secure storage and for excluding it from any version control system.

3.3.9. Auxiliary Scripts

Two scripts were developed to support the deployment and testing of the AQDR platform.

- The first is a data migration utility that loads historical air quality readings, exported as CSV files from any of the project's pilots, into InfluxDB. It maps each file to its

corresponding sensor via the PostgreSQL sensors table and reshapes the readings to match the format expected by the platform's ingestor. Full details and usage examples are provided in Appendix I: CSV to InfluxDB Historical Data Loader Script.

The second is a lightweight mock server that emulates the three external sensor APIs consumed by the AQDR Ingestor — Inbiot, Qlair and Kunak. It allows the platform to be developed and tested locally, generating realistic random readings without relying on the real external services, and can be started or stopped independently of the rest of the stack. Its implementation and configuration are described in Appendix II: Mock Sensor APIs Server.

4. Conclusions

This deliverable reports on the final milestone of the Data Management Module's development, focusing on the handover of the Air Quality Data Repository (AQDR) submodule to the project coordinator and ensuring the platform's continuity beyond the project's active development period.

Besides minor software updates and data recovery from malfunctioning sensors, the core outcome of this work has been the successful transfer of both the AQDR source code and its deployment mechanism. To remove the dependency on Kubernetes infrastructure, an alternative, self-contained deployment package based on Docker Compose was designed, implemented, and validated, allowing the AQDR to be installed and operated on a single Linux virtual machine without requiring familiarity with container orchestration. This package faithfully reproduces the functional behaviour of the production environment (same container images, configuration structure, and health-check logic) while replacing Kubernetes-specific constructs with their Docker Compose equivalents.

Beyond an empty deployment capability, a dedicated restore procedure was developed and executed to migrate the existing, populated production instance (the sensor registry and the full historical air quality time series) into the coordinator's new environment, completing a live handover rather than a clean-slate installation. This process was documented in detail as a practical operational guide, including prerequisites, directory structure, configuration, first-time deployment, and data restore steps, so that platform administrators can operate the AQDR independently going forward.

Finally, two auxiliary scripts were developed to reduce the operational overhead of working with the platform's data pipeline: a CSV-to-InfluxDB loader that enables the onboarding of historical pilot data into the platform's data model, and a mock sensor API server that removes the dependency on external providers during development and testing. Together with the handover package itself, these tools support a sustainable, reproducible, and provider-independent operation of the AQDR beyond the end of the project.

As agreed within the project, the DAQSR and HDH submodules remain deployable only through their original Kubernetes-based descriptors. Extending the simplified Docker Compose mechanism to cover these components, along with the ongoing operational ownership of the platform by the project coordinator, remain a future post-project activity if required, outside the scope of this deliverable.

Appendix I: CSV to InfluxDB Historical Data Loader Script

This script (*load_csv_to_influx.py*) automates the loading of historical air quality data, exported in CSV format from any of the project's pilots, into the InfluxDB database. The script performs the following functions:

- Reads CSV files following the naming pattern data_ *<scenario><area>.csv*.
- Maps each file to its corresponding sensor using the exported PostgreSQL sensors table.
- Transforms the IAQ columns (temperature, humidity, CO2, formaldehyde, TVOC, PM1, PM10, PM2.5) to match the data model used by the InfluxDB ingestor.
- Writes the resulting points to InfluxDB in batches of 5000 points.
- Automatically generates the SQL statements needed to update the inclusion time field for each sensor in PostgreSQL, based on the earliest date found in the data.

The script supports a DRY_RUN mode to simulate execution without writing actual data, which is useful for validating the sensor mapping and data format before the final load.

```
→Connection variables:

$export INFLUXDB_URL="http://localhost:8086"
$export DOCKER_INFLUXDB_INIT_ADMIN_TOKEN="your-influxdb-token"
$export DOCKER_INFLUXDB_INIT_ORG="your-org"
$export DOCKER_INFLUXDB_INIT_BUCKET="AIR_QUALITY"

→Custom data files

$export CSV_DIR="/data/pilot_exports/csv_data"
$export SENSORS_TABLE="/data/pilot_exports/sensors_table.csv"

→Script execution

$python load_csv_to_influx.py

→DryRun Script execution

$export DRY_RUN=true
$python load_csv_to_influx.py
```

Figure 5: K-HEALTHinAIR CSV data loader examples of use data restoring.

```
"""
Load inBiot CSV historical data into InfluxDB.

Reads CSV files exported by inBiot (format: data_<scenario><area>.csv),
maps them to the correct sensor using the PostgreSQL sensors table export,
transforms IAQ columns to match the ingestor's InfluxDB data model,
and writes the points to InfluxDB.

Usage:
    1. Place all data_*.csv files in a folder (e.g. ./csv_data/)
    2. Place the PostgreSQL sensors table export as sensors_table.csv
    3. Set environment variables (or edit the CONFIG section below)
    4. Run: python load_csv_to_influx.py

Environment variables (or edit CONFIG below):
    INFLUXDB_URL                        e.g. http://localhost:8086
    DOCKER_INFLUXDB_INIT_ADMIN_TOKEN    InfluxDB admin token
    DOCKER_INFLUXDB_INIT_ORG            InfluxDB org
    DOCKER_INFLUXDB_INIT_BUCKET         e.g. AIR_QUALITY
"""

import csv
import glob
import os
import re
import sys
from datetime import datetime, timezone

from influxdb_client import InfluxDBClient, Point
from influxdb_client.client.write_api import SYNCHRONOUS

# ============================================================================
# CONFIG — edit these or set as environment variables
# ============================================================================
INFLUXDB_URL = os.getenv("INFLUXDB_URL", "http://localhost:8086")
INFLUXDB_TOKEN = os.getenv("DOCKER_INFLUXDB_INIT_ADMIN_TOKEN", "")
INFLUXDB_ORG = os.getenv("DOCKER_INFLUXDB_INIT_ORG", "")
INFLUXDB_BUCKET = os.getenv("DOCKER_INFLUXDB_INIT_BUCKET", "AIR_QUALITY")

CSV_DIR = os.getenv("CSV_DIR", "./csv_data")          # folder with data_*.csv
SENSORS_TABLE = os.getenv("SENSORS_TABLE", "./sensors_table.csv")  # pg export
MEASUREMENT = "CONTINUOUS_MONITORING_"
BATCH_SIZE = 5000   # points per write batch
DRY_RUN = os.getenv("DRY_RUN", "false").lower() == "true"  # set true to test

# ============================================================================
# CSV column -> InfluxDB field mapping (matching ingestor ind_linkage)
# ============================================================================
# CSV columns use ";" separator, latin1 encoding
# We map by column position (0-indexed) to avoid encoding issues in headers
#
#  Col 0: Date           (dd/mm/yyyy)
#  Col 1: Time           (HH:MM:SS)
#  Col 2: Ventilation Indicator   -- SKIP (not in data model)
#  Col 3: IAQ Indicator           -- SKIP
#  Col 4: Virus Index             -- SKIP
#  Col 5: thermalIndicator        -- SKIP
#  Col 6: Temperature (°C)        -> temperature
#  Col 7: Humidity (%)             -> humidity
#  Col 8: CO2 (ppm)               -> co2
#  Col 9: Formaldehyde (µg/m³)    -> formaldehyde
#  Col 10: TVOC (ppb)             -> tvocs
#  Col 11: PM 1.0 (µg/m³)         -> pm1
#  Col 12: PM 4.0 (µg/m³)         -- SKIP (not in data model)
#  Col 13: PM 10 (µg/m³)          -> pm10
```

```
#  Col 14: PM 2.5 (µg/m³)         -> pm25
#  Col 15-20: Weather/Outdoor     -- SKIP (different dates, not IAQ)

COL_MAPPING = {
    6:  "temperature",
    7:  "humidity",
    8:  "co2",
    9:  "formaldehyde",
    10: "tvocs",
    11: "pm1",
    13: "pm10",
    14: "pm25",
}


def load_sensors_table(path):
    """Load PostgreSQL sensors table export and build lookup by scenario+area.

    Returns dict: { "RES0411": { sensor_id, pilot, scenario, sub_scenario,
                                 area_tag, ... }, ... }
    """
    sensors = {}
    with open(path, "r", encoding="utf-8-sig") as f:
        reader = csv.DictReader(f)
        for row in reader:
            # Clean possible quotes and whitespace
            sensor_id = row["sensor_id"].strip().strip('"')
            scenario_full = row["scenario"].strip()   # e.g. "RES01"
            area_num = row["area"].strip()             # e.g. "11"

            # Key matching filename pattern: data_RES0411.csv
            key = f"{scenario_full}{area_num}"

            # Tags exactly as the ingestor builds them
            scenario_tag = scenario_full[:3]           # "RES"
            sub_scenario_tag = scenario_full           # "RES01"
            area_tag = f"{scenario_full}{str(int(area_num)).zfill(3)}"  # "RES01011"
            pilot = row["pilot"].strip()               # "NO"

            sensors[key] = {
                "sensor_id": sensor_id,
                "pilot": pilot,
                "scenario": scenario_tag,
                "sub_scenario": sub_scenario_tag,
                "area": area_tag,
                "scenario_full": scenario_full,
                "area_num": area_num,
            }
    return sensors


def parse_csv_row_to_fields(cols):
    """Extract IAQ fields from a CSV row (list of column values).

    Returns (timestamp_utc, fields_dict) or (None, None) if row is invalid.
    """
    # Parse date + time -> UTC timestamp
    date_str = cols[0].strip()  # dd/mm/yyyy
    time_str = cols[1].strip()  # HH:MM:SS

    if not date_str or not time_str:
        return None, None

    try:
        # Support both date formats: dd/mm/yyyy and yyyy-mm-dd
        if "/" in date_str:
```

```
            dt = datetime.strptime(f"{date_str} {time_str}", "%d/%m/%Y %H:%M:%S")
        else:
            dt = datetime.strptime(f"{date_str} {time_str}", "%Y-%m-%d %H:%M:%S")
        dt = dt.replace(tzinfo=timezone.utc)
        ts_epoch = int(dt.timestamp())
    except ValueError:
        return None, None

    # Extract fields
    fields = {}
    for col_idx, field_name in COL_MAPPING.items():
        if col_idx < len(cols) and cols[col_idx].strip():
            try:
                value = float(cols[col_idx].strip())
                fields[field_name] = round(value, 2)
            except ValueError:
                continue

    if not fields:
        return None, None

    return ts_epoch, fields


def build_point(ts_epoch, fields, sensor_info):
    """Build an InfluxDB Point matching the ingestor's format."""
    dp = {
        "measurement": MEASUREMENT,
        "tags": {
            "AQ": "IAQ",
            "id_pilot": sensor_info["pilot"],
            "scenario": sensor_info["scenario"],
            "sub_scenario": sensor_info["sub_scenario"],
            "area": sensor_info["area"],
            "sensor_id": sensor_info["sensor_id"],
        },
        "fields": fields,
        "time": ts_epoch,
    }
    return Point.from_dict(dp, write_precision="s")


def process_csv(csv_path, sensor_info, writer):
    """Process a single CSV file and write points to InfluxDB.

    Returns (points_written, first_date, last_date).
    """
    points = []
    first_ts = None
    last_ts = None
    skipped = 0

    # Read with latin1 encoding (inBiot exports use ISO-8859-1)
    with open(csv_path, "r", encoding="latin1") as f:
        reader = csv.reader(f, delimiter=";")
        header = next(reader)  # skip header

        for row_num, row in enumerate(reader, start=2):
            if len(row) < 15:
                skipped += 1
                continue

            ts_epoch, fields = parse_csv_row_to_fields(row)
            if ts_epoch is None:
                skipped += 1
                continue
```

```
            if first_ts is None or ts_epoch < first_ts:
                first_ts = ts_epoch
             if last_ts is None or ts_epoch > last_ts:
                last_ts = ts_epoch

             point = build_point(ts_epoch, fields, sensor_info)
             points.append(point)

             # Write in batches
             if len(points) >= BATCH_SIZE:
                 if not DRY_RUN:
                     writer.write(
                         bucket=INFLUXDB_BUCKET,
                         org=INFLUXDB_ORG,
                         record=points,
                     )
                 print(f"    ... wrote batch of {len(points)} points")
                 points = []

    # Write remaining points
    if points:
        if not DRY_RUN:
            writer.write(
                bucket=INFLUXDB_BUCKET,
                org=INFLUXDB_ORG,
                record=points,
            )

    first_dt = datetime.fromtimestamp(first_ts, timezone.utc) if first_ts else None
    last_dt = datetime.fromtimestamp(last_ts, timezone.utc) if last_ts else None

    total = (row_num - 1) if 'row_num' in dir() else 0
    written = total - skipped

    return written, skipped, first_dt, last_dt


def main():
    print("=" * 70)
    print("inBiot CSV -> InfluxDB Loader")
    print("=" * 70)

    if DRY_RUN:
        print("*** DRY RUN MODE - no data will be written ***\n")

    # Load sensor mapping
    print(f"Loading sensors table from: {SENSORS_TABLE}")
    sensors = load_sensors_table(SENSORS_TABLE)
    print(f"  Found {len(sensors)} sensors\n")

    # Find CSV files
    csv_files = sorted(glob.glob(os.path.join(CSV_DIR, "data_*.csv")))
    if not csv_files:
        print(f"ERROR: No data_*.csv files found in {CSV_DIR}")
        sys.exit(1)
    print(f"Found {len(csv_files)} CSV files in {CSV_DIR}\n")

    # Connect to InfluxDB
    if not DRY_RUN:
        print(f"Connecting to InfluxDB at {INFLUXDB_URL}")
        client = InfluxDBClient(
            url=INFLUXDB_URL, token=INFLUXDB_TOKEN, org=INFLUXDB_ORG
        )
        writer = client.write_api(write_options=SYNCHRONOUS)
        print("  Connected.\n")
    else:
        client = None
```

```
        writer = None

    # SQL statements for inclusion_time updates
    sql_updates = []

    # Process each CSV
    for csv_path in csv_files:
        filename = os.path.basename(csv_path)

        # Extract scenario+area from filename: data_RES0411.csv -> RES0411
        match = re.match(r"data_([A-Z]+\d+)\.csv", filename)
        if not match:
            print(f"SKIP {filename}: cannot parse scenario+area from name")
            continue

        raw_key = match.group(1)  # e.g. "RES01011"

        # Match against sensors table, trying multiple interpretations
        # Filenames may use zero-padded area (RES01011 = RES01 + 011)
        # while the sensors table key uses unpadded area (RES01 + 11 = RES0111)
        key = None
        if raw_key in sensors:
            key = raw_key
        else:
            # Try scenario (first 5 chars) + area (remaining, stripped of
            # leading zeros): RES01011 -> RES01 + 011 -> RES01 + 11 -> RES0111
            for scenario_len in [5, 4]:  # try RES01 (5) then RES0 (4)
                if len(raw_key) > scenario_len:
                    scenario_part = raw_key[:scenario_len]
                    area_part = raw_key[scenario_len:].lstrip("0") or "0"
                    candidate = scenario_part + area_part
                    if candidate in sensors:
                        key = candidate
                        break

        if key is None:
            print(f"SKIP {filename}: no sensor found for key '{raw_key}'")
            print(f"  Available keys: {sorted(sensors.keys())}")
            continue

        sensor_info = sensors[key]
        sid_short = sensor_info["sensor_id"][:8]

        print(f"Processing {filename}")
        print(f"  -> sensor_id: {sid_short}... "
              f"({sensor_info['sub_scenario']} area {sensor_info['area_num']})")

        try:
            written, skipped, first_dt, last_dt = process_csv(
                csv_path, sensor_info, writer
            )
        except Exception as e:
            print(f"  ERROR: {e}")
            continue

        print(f"  -> {written} points written, {skipped} skipped")
        if first_dt:
            print(f"  -> Date range: {first_dt:%Y-%m-%d %H:%M} to "
                  f"{last_dt:%Y-%m-%d %H:%M} UTC")

            # Generate SQL update for inclusion_time
            sql = (
                f"UPDATE {sensors_table_name()} "
                f"SET inclusion_time = '{first_dt:%Y-%m-%d %H:%M:%S}' "
                f"WHERE sensor_id = '{sensor_info['sensor_id']}';"
            )
            sql_updates.append((filename, sensor_info["sensor_id"][:8], sql))
```

```
        print()

    # Print SQL updates
    if sql_updates:
        print("=" * 70)
        print("SQL: UPDATE inclusion_time for each sensor")
        print("=" * 70)
        for fname, sid, sql in sql_updates:
            print(f"-- {fname} ({sid}...)")
            print(sql)
        print()

        # Also write to file
        sql_path = os.path.join(
            os.path.dirname(SENSORS_TABLE), "update_inclusion_time.sql"
        )
        with open(sql_path, "w") as f:
            f.write("-- Auto-generated: update inclusion_time based on CSV data\n")
            f.write(f"-- Generated: {datetime.now():%Y-%m-%d %H:%M:%S}\n\n")
            for fname, sid, sql in sql_updates:
                f.write(f"-- {fname} ({sid}...)\n{sql}\n\n")
        print(f"SQL file written to: {sql_path}")

    if not DRY_RUN and client:
        client.close()

    print("\nDone.")


def sensors_table_name():
    """Return the PostgreSQL table name. Adjust if needed."""
    return "sensors"


if __name__ == "__main__":
    main()
```

Figure 6: K-HEALTHinAIR CSV data loader script.

Appendix II: Mock Sensor APIs Server

This standalone mock server (*mock_server.py*) simulates the three external sensor APIs used by the AQDR Ingestor, enabling local development and testing without depending on the real external services. It is deployed independently from the AQDR stack, so it can be started and stopped without affecting the rest of the platform.

```
mock-sensor-apis/
├── docker-compose.yaml     # Standalone compose
├── .env                    # Mock configuration
├── Dockerfile              # Image based on python:3.11-slim
├── mock_server.py          # FastAPI server with the three mocks
└── README.md
```

Figure 7: K-HEALTHinAIR sensor server mock structure.

Table 5: Sensor server mock end points

API	Method	Endpoint	Description
Inbiot	GET	/last-measurements/{apikey}/{sensor_id}	Latest indoor reading
Inbiot	GET	/measurements-by-time/{apikey}/{sensor_id}/{init}/{end}	Indoor historical data
Qlair	GET	/open/indoor_air_quality?mac=…&start=…&end=…	Indoor (latest or historical)
Qlair	GET	/open/outdoor_air_quality?building_id=…&start=…&end=…	Outdoor (latest or historical)
Kunak	POST	/rest/devices/{id}/reads/fromGroupByTs	Readings from a given timestamp
Kunak	POST	/rest/devices/{id}/reads/fromToGroupByTs	Readings between two timestamps

Each mock generates random but realistic values within predefined ranges for its corresponding magnitudes (temperature, humidity, CO2, TVOC, particulate matter, formaldehyde, NO2, pressure, etc.).

Configuration (*.env*):

Table 6: Mock connection parameters

Variable	Default	Description
MOCK_PORT	8888	Mock server port
MOCK_NUM_READINGS	24	Readings per historical request
MOCK_INTERVAL_MIN	60	Minutes between generated readings

For load testing, these can be adjusted, e.g. *MOCK_NUM_READINGS=1000* and *MOCK_INTERVAL_MIN=5*, to generate a larger volume of readings at a higher frequency.

Integration with AQDR: once the mock is running, the three API URLs in the AQDR *.env* file are pointed to the mock server (http://mock-sensor-apis:8888) instead of the real endpoints, and the services that consume them (*aqdr-api, aqdr-ingestor*) are restarted. Communication works because both stacks share the *khealthinair-net* Docker network.

```
cd ~/khealthinair-mock-sensor-apis

# Start
docker-compose up -d

# Check health
curl http://127.0.0.1:8888/health
curl http://127.0.0.1:8888/

# View logs
docker-compose logs -f

# Stop
docker-compose down
```

Figure 8: K-HEALTHinAIR sensor server mock example of use.

Once running, update the AQDR .env to point the three API URLs to the mock server, then restart the dependent services:

```
cd ~/khealthinair-aqdr/deployment/docker-compose
docker-compose restart aqdr-api aqdr-ingestor
```

Figure 9: K-HEALTHinAIR sensor server mock example of use II.

```
"""
K-HEALTHinAIR — Mock API Server for Inbiot, Qlair & Kunak
==========================================================
Single FastAPI server that simulates the three external sensor APIs.
Generates random but realistic air quality readings.

Configuration via environment variables:
  MOCK_NUM_READINGS   — number of readings per historic request (default: 24)
  MOCK_INTERVAL_MIN   — minutes between readings (default: 60)
  MOCK_PORT           — port to listen on (default: 8888)
"""

import os
import random
import time
from datetime import datetime, timezone
from typing import Optional

from fastapi import FastAPI, Header, Query, Request
from fastapi.responses import JSONResponse
import uvicorn

app = FastAPI(title="K-HEALTHinAIR Mock Sensor APIs")

# ---------------------------------------------------------------------------
# Configuration
# ---------------------------------------------------------------------------
NUM_READINGS = int(os.getenv("MOCK_NUM_READINGS", "24"))
INTERVAL_MIN = int(os.getenv("MOCK_INTERVAL_MIN", "60"))

# ---------------------------------------------------------------------------
# Helpers — realistic random ranges
# ---------------------------------------------------------------------------
INDOOR_RANGES = {
    "temperature": (18.0, 28.0),
    "humidity": (30.0, 70.0),
    "co2": (350.0, 1200.0),
    "tvocs": (0.0, 500.0),
    "pm1": (0.0, 15.0),
    "pm10": (0.0, 50.0),
    "pm25": (0.0, 35.0),
    "formaldehyde": (0.0, 50.0),
}

OUTDOOR_RANGES = {
    "temperature": (-10.0, 35.0),
    "humidity": (20.0, 95.0),
    "pm1": (0.0, 10.0),
    "pm25": (0.0, 30.0),
    "pm10": (0.0, 60.0),
    "no2": (0.0, 80.0),
    "o3": (0.0, 120.0),
    "co": (0.0, 5.0),
    "so2": (0.0, 10.0),
    "vocs": (0.0, 5.0),
    "pressure": (990.0, 1035.0),
}
def _rv(lo: float, hi: float) -> float:
    return round(random.uniform(lo, hi), 2)


def _now_ms() -> int:
    return int(time.time() * 1000)


def _now_s() -> int:
    return int(time.time())
```

```
# ================================================================
# INBIOT MOCK — https://docs.myinbiotpublicapi.com
# ================================================================

def _inbiot_measurement(ts_ms: int) -> list:
    """Build systemData array for one timestamp."""
    types = [
        ("temperature", "CELSIUS"),
        ("humidity", "%"),
        ("co2", "ppm"),
        ("vocs", "ppb"),
        ("pm1", "µg/m³"),
        ("pm10", "µg/m³"),
        ("pm25", "µg/m³"),
        ("formaldehyde", "ppb"),
    ]
    system_data = []
    for idx, (t, unit) in enumerate(types):
        lo, hi = INDOOR_RANGES.get(t, (0, 100))
        system_data.append({
            "type": t,
            "_id": idx,
            "settings": {
                "lastMeasureDate": datetime.fromtimestamp(
                    ts_ms / 1000, tz=timezone.utc
                ).strftime("%Y-%m-%dT%H:%M:%S.000Z"),
                "measured": True,
                "rangeUnit": "h",
                "rangeValue": 24,
                "measurementUnit": unit,
            },
            "measurements": [{"value": _rv(lo, hi), "date": ts_ms}],
        })
    return system_data


def _inbiot_measurement_hist(num: int) -> list:
    """Build systemData array with multiple measurements per type."""
    now = _now_ms()
    types = [
        ("temperature", "CELSIUS"),
        ("humidity", "%"),
        ("co2", "ppm"),
        ("vocs", "ppb"),
        ("pm1", "µg/m³"),
        ("pm10", "µg/m³"),
        ("pm25", "µg/m³"),
        ("formaldehyde", "ppb"),
    ]
    system_data = []
    for idx, (t, unit) in enumerate(types):
        lo, hi = INDOOR_RANGES.get(t, (0, 100))
        measurements = []
        for i in range(num):
            ts = now - (i * INTERVAL_MIN * 60 * 1000)
            measurements.append({"value": _rv(lo, hi), "date": ts})
        system_data.append({
            "type": t,
            "_id": idx,
            "settings": {
                "lastMeasureDate": datetime.fromtimestamp(
                    now / 1000, tz=timezone.utc
                ).strftime("%Y-%m-%dT%H:%M:%S.000Z"),
                "measured": True,
                "rangeUnit": "h",
```

```
                "rangeValue": 24,
                "measurementUnit": unit,
            },
            "measurements": measurements,
        })
    return system_data


@app.get("/last-measurements/{apikey}/{sensor_id}")
async def inbiot_last(apikey: str, sensor_id: str):
    """Inbiot: latest measurements for a sensor."""
    return {"systemData": _inbiot_measurement(_now_ms())}


@app.get("/measurements-by-time/{apikey}/{sensor_id}/{init_dt}/{end_dt}")
async def inbiot_hist(apikey: str, sensor_id: str, init_dt: str, end_dt: str):
    """Inbiot: historical measurements for a time range."""
    return {"systemData": _inbiot_measurement_hist(NUM_READINGS)}


# ================================================================
# QLAIR MOCK — https://api.v2.i-qlair.eu/docs
# ================================================================

def _qlair_indoor_record(ts_s: float) -> dict:
    """Build one indoor record."""
    return {
        "pmtwoaqi": 0.0,
        "eventdatetime": ts_s,
        "chemisense": {
            "total_aqi": random.randint(10, 80),
            "productivity_score": _rv(70, 100),
            "covid": _rv(50, 100),
            "voc_ppb": _rv(*INDOOR_RANGES["tvocs"]),
            "comfort_score": _rv(70, 100),
            "cotwo": _rv(*INDOOR_RANGES["co2"]),
            "h": _rv(*INDOOR_RANGES["humidity"]),
            "voc": _rv(0, 10),
            "total_aqi_hundred": _rv(80, 100),
            "ventilation_score": _rv(5, 60),
            "air_cost": _rv(400, 900),
            "asthma_score": _rv(80, 100),
            "pmtwo": _rv(*INDOOR_RANGES["pm25"]),
            "mold_score": _rv(80, 100),
            "t": _rv(*INDOOR_RANGES["temperature"]),
            "pmone": _rv(*INDOOR_RANGES["pm1"]),
            "pmten": _rv(*INDOOR_RANGES["pm10"]),
            "sensedtimedate": ts_s,
            "Total_AQI": random.randint(10, 80),
        },
        "mac": "mock_sensor_mac_address",
        "pmtenaqi": 0.0,
    }


def _qlair_outdoor_record(ts_s: float, building_id: int) -> dict:
    """Build one outdoor record."""
    return {
        "aqi": {
            "category": "Good",
            "type": "airnow",
            "value": float(random.randint(10, 80)),
        },
        "eventdatetime": ts_s,
        "facilityname": str(building_id),
```

```
        "pollutants": {
            "Total_AQI": random.randint(10, 80),
            "co": _rv(*OUTDOOR_RANGES["co"]),
            "h": _rv(*OUTDOOR_RANGES["humidity"]),
            "no2": _rv(*OUTDOOR_RANGES["no2"]),
            "o3": _rv(*OUTDOOR_RANGES["o3"]),
            "pm10": _rv(*OUTDOOR_RANGES["pm10"]),
            "pm25": _rv(*OUTDOOR_RANGES["pm25"]),
            "pmten": _rv(*OUTDOOR_RANGES["pm10"]),
            "pmtwo": _rv(*OUTDOOR_RANGES["pm25"]),
            "so2": _rv(*OUTDOOR_RANGES["so2"]),
            "t": _rv(*OUTDOOR_RANGES["temperature"]),
            "total_aqi": random.randint(10, 80),
        },
        "sensedtimedate": int(ts_s),
        "weather": {
            "h": _rv(*OUTDOOR_RANGES["humidity"]),
            "t": _rv(*OUTDOOR_RANGES["temperature"]),
        },
    }


@app.get("/open/indoor_air_quality")
async def qlair_indoor(
    mac: str = Query(None),
    start: Optional[int] = Query(None),
    end: Optional[int] = Query(None),
    x_api_key: Optional[str] = Header(None),
):
    """Qlair: indoor air quality — latest or historic."""
    if start and end:
        records = []
        ts = start
        while ts <= end:
            records.append(_qlair_indoor_record(float(ts)))
            ts += INTERVAL_MIN * 60
        return records
    return _qlair_indoor_record(float(_now_s()))


@app.get("/open/outdoor_air_quality")
async def qlair_outdoor(
    building_id: int = Query(91),
    start: Optional[int] = Query(None),
    end: Optional[int] = Query(None),
    x_api_key: Optional[str] = Header(None),
):
    """Qlair: outdoor air quality — latest or historic."""
    if start and end:
        records = []
        ts = start
        while ts <= end:
            records.append(_qlair_outdoor_record(float(ts), building_id))
            ts += INTERVAL_MIN * 60
        return records
    return _qlair_outdoor_record(float(_now_s()), building_id)


# ═══════════════════════════════════════════════════════════════
# KUNAK MOCK — https://kunakcloud.com/openAPIv0/v1
# ═══════════════════════════════════════════════════════════════

def _kunak_record(ts_ms: int) -> dict:
    """Build one Kunak outdoor record."""
    return {
```

```
        "ts": ts_ms,
        "PM1": {"value": str(_rv(*OUTDOOR_RANGES["pm1"])), "validation": "T", "reason": "0"},
        "PM2.5": {"value": str(_rv(*OUTDOOR_RANGES["pm25"])), "validation": "T", "reason": "0"},
        "PM10": {"value": str(_rv(*OUTDOOR_RANGES["pm10"])), "validation": "T", "reason": "0"},
        "Temp  ext":  {"value":  str(_rv(*OUTDOOR_RANGES["temperature"])),  "validation":  "T",
"reason": "0"},
        "Humidity  ext":  {"value":  str(_rv(*OUTDOOR_RANGES["humidity"])),  "validation":  "T",
"reason": "0"},
        "NO2 GCc": {"value": str(_rv(*OUTDOOR_RANGES["no2"])), "validation": "T", "reason": "0"},
        "VOCs  GCc":  {"value":  str(_rv(*OUTDOOR_RANGES["vocs"])),  "validation":  "T",  "reason":
"0"},
        "Pressure":   {"value":   str(_rv(*OUTDOOR_RANGES["pressure"])),   "validation":   "T",
"reason": "0"},
    }


@app.post("/rest/devices/{device_id}/reads/fromGroupByTs")
async def kunak_from(device_id: str, request: Request):
    """Kunak: reads from a timestamp onwards."""
    records = []
    now = _now_ms()
    for i in range(NUM_READINGS):
        ts = now - (i * INTERVAL_MIN * 60 * 1000)
        records.append(_kunak_record(ts))
    records.sort(key=lambda r: r["ts"])
    return records


@app.post("/rest/devices/{device_id}/reads/fromToGroupByTs")
async def kunak_from_to(device_id: str, request: Request):
    """Kunak: reads between two timestamps."""
    records = []
    now = _now_ms()
    for i in range(NUM_READINGS):
        ts = now - (i * INTERVAL_MIN * 60 * 1000)
        records.append(_kunak_record(ts))
    records.sort(key=lambda r: r["ts"])
    return records


# ================================================================
# Health & info
# ================================================================

@app.get("/health")
async def health():
    return {"status": "ok", "service": "mock-sensor-apis"}


@app.get("/")
async def root():
    return {
        "service": "K-HEALTHinAIR Mock Sensor APIs",
        "endpoints": {
            "inbiot": [
                "/last-measurements/{apikey}/{sensor_id}",
                "/measurements-by-time/{apikey}/{sensor_id}/{init}/{end}",
            ],
```

```
            "qlair": [
                "/open/indoor_air_quality?mac=...&start=...&end=...",
                "/open/outdoor_air_quality?building_id=...&start=...&end=...",
            ],
            "kunak": [
                "/rest/devices/{id}/reads/fromGroupByTs",
                "/rest/devices/{id}/reads/fromToGroupByTs",
            ],
        },
        "config": {
            "MOCK_NUM_READINGS": NUM_READINGS,
            "MOCK_INTERVAL_MIN": INTERVAL_MIN,
        },
    }


if __name__ == "__main__":
    port = int(os.getenv("MOCK_PORT", "8888"))
    uvicorn.run(app, host="0.0.0.0", port=port)
```

Figure 10: K-HEALTHinAIR sensor server mock script.