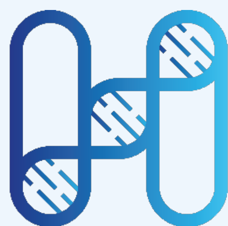


HEREDITARY

HetERogeneous sEmantic Data integratIon for the guT-bRain interplaY

Deliverable 3.2

Federated workflow execution methods: first release

Version 2.80, 23 December 2025

This project has received funding from the European Union's Horizon Europe research and innovation programme under grant agreement No GA 101137074. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union. Neither the European Union nor the granting authority can be held responsible for them.

Funded by the European Union

EXECUTIVE SUMMARY

This deliverable introduces the *Hereditary Data Network (HDN)*, a privacy-by-design federation architecture for medical data analytics within the HEREDITARY project. HDN addresses the need to perform cross-institutional analyses and model development over sensitive clinical, genomic, and imaging data without centralizing patient-level information, in compliance with regulations such as the GDPR, the Data Governance Act, and the emerging AI Act.

HDN provides a unified semantic view of consortium data through the *Hereditary Ontology (HERO)* ontology and an ontology-mediated query interface. Researchers express information needs as SPARQL queries over a stable conceptual schema, while participating institutions retain full control over storage technologies, local schemas, and disclosure policies. Architecturally, HDN follows a hub-and-spoke model: a central orchestrator manages a vetted catalog of query templates, validates requests, enforces disclosure-level constraints at the semantic boundary, and dispatches instantiated queries to institutional endpoints. Each endpoint operates an *Ontology-Based Data Access (OBDA)*-based stack that maps ontology-level queries to its local schema, executes them under local privacy rules, and returns only admissible aggregated or record-level results.

The deliverable makes four main contributions. First, it formalizes the evolution from the initial *Ontology-Based Data Federation (OBDF)* architecture to a native federation design that embeds privacy enforcement into the core protocol, rather than as an external layer. Second, it details the logical and reference implementations of HDN Central and HDN Endpoints, including interaction protocols, query lifecycle, and privacy controls. Third, it presents a benchmark comparing HDN against the legacy OBDF approach, showing improved scalability, more robust behavior as the number of endpoints grows, and better alignment with institutional privacy constraints. Finally, it demonstrates the applicability of HDN through three use cases: (1) federated queries on ALS clinical data at different disclosure levels, (2) a distributed SQL-based implementation of a machine learning algorithm, the Cox survival model, and (3) integration with Ontotext's LinkedLifeData Inventory for FAIR-compliant external datasets. Together, these results show that HDN provides a practical and extensible foundation for federated analytics in HEREDITARY and prepares the ground for tighter integration with federated learning workflows in future project phases.

DOCUMENT INFORMATION

Deliverable ID	3.2
Deliverable Title	Federated workflow execution methods: first release
Work Package	WP 3
Lead Partner	Aalborg University
Due Date	31 December 2025
Date of submission	23 December 2025
Deliverable Type	R – Report
Dissemination level	PU – Public

AUTHORS

Name	Organization
Maria Barouh	ONTO
Svetla Boytcheva	ONTO
Mirco Cazzaro	UNIPD
Daniele Dell'Aglio	AAU
Luca Fabbian	AAU
Pavlin Gyurov	ONTO
Juan Manuel Rodriguez	AAU
Gianmaria Silvello	UNIPD

REVISION HISTORY

Version	Date	Author	Description
0.10	01.10.2025	Daniele Dell'Aglio	Initial table of content
0.20	03.10.2025	Mirco Cazzaro	Draft of sections about HDN
0.30	14.10.2025	All	Revision of the document structure and refinement of the table of contents
0.40	19.10.2025	Juan Manuel Rodriguez	First draft of the introduction
0.50	19.10.2025	Daniele Dell'Aglio	First part of the HDN section
0.60	26.10.2025	Luca Fabbian	First draft of the Cox case study
0.70	31.10.2025	Mirco Cazzaro	Section 2: OBDF experiment setup and results
0.80	11.11.2025	Mirco Cazzaro	Section 2: HDN experiment setup and results
0.90	12.11.2025	Juan Manuel Rodriguez	Review on the Cox case study
1.00	16.11.2025	Gianmaria Silvello	Annotations and revision of Section 2
1.10	16.11.2025	Daniele Dell'Aglio	Revised the introduction of the HDN section
1.20	20.11.2025	Mirco Cazzaro	Section 3: ALS use case with HDN results
1.30	22.11.2025	Luca Fabbian	Work on Cox Case Study
1.40	24.11.2025	Maria Barouh and Pavlin Gyurov	Work on section 3.3 (LLDI and selected datasets)
1.50	23.11.2025	Mirco Cazzaro	Applying Sections 2 and 3 revisions
1.60	25.11.2025	Daniele Dell'Aglio	Revision of the deliverable
2.00	25.11.2025	Juan Manuel Rodriguez	Revision and submitted for internal review
2.10	01.12.2025	Gianmaria Silvello	Full Revision
2.20	08.12.2025	Luca Fabbian	Added the list of acronyms, updated the bibliography entries and reviewed the conclusions
2.30	10.12.2025	Maria Barouh	Revised Section 3.3
2.40	10.12.2025	Luca Fabbian	Reworked the introduction and scope of Section 3.2
2.50	16.12.2025	Mirco Cazzaro	Revised the sections describing HDN
2.60	10.12.2025	Luca Fabbian	Revised Section 3.2
2.70	22.12.2025	Maria Barouh and Pavlin Gyurov	Revised Section 3.3
2.75	22.12.2025	AAU Team	Rivision of figures and references
2.80	23.12.2025	Gianmaria Silvello	Full Rivision and minor fixes

HEREDITARY

Funded by the European Union

Contents

1 Introduction **11**
1.1 Contributions . . . 11

2 The Hereditary Data Network **12**
2.1 Architectural Evolution: From Centralized Middleware to Native Federation . . . 13
2.1.1 Principle: Privacy by Design . . . 15
2.2 HDN Architecture: Components and Interactions . . . 16
2.2.1 Architectural Components . . . 16
2.2.2 Privacy Levels . . . 16
2.2.3 Interaction Protocol . . . 17
2.3 HDN Central: the HDN Federation Engine . . . 19
2.3.1 Design Principles . . . 19
2.3.2 Logical Architecture of HDN Central . . . 19
2.3.3 Federated Query Execution . . . 21
2.4 HDN Endpoints: the Local HDN Nodes . . . 21
2.4.1 Role and Design Principles . . . 22
2.4.2 Logical Architecture . . . 22
2.4.3 Architecture and Data Pipeline of the Reference HDN Endpoint . . . 22
2.4.4 Querying and Privacy Control . . . 24
2.5 Comparison of the Previous Architecture with the New Architecture . . . 24
2.5.1 Experimental Benchmark . . . 24
2.5.2 Experimental Setup . . . 25
2.5.3 OBDF Baseline Results . . . 27
2.5.4 HDN Results . . . 27
2.5.5 Comparative Analysis . . . 27
2.6 Implementation Notes . . . 32
2.6.1 Reference Implementation of HDN Central . . . 33
2.6.2 Reference Implementation of HDN Endpoints . . . 33

3 Use Case Studies **34**
3.1 Query Distribution on ALS Data . . . 34
3.1.1 Local Data: Quantity and Characteristics . . . 34
3.1.2 Query Templates and Privacy Levels . . . 35
3.1.3 Use case: Federated Average Age at Onset for ALS (L2) . . . 36
3.2 Towards Machine Learning in HDN: a Case Study on the Cox model . . . 38
3.2.1 Background on the Cox Model . . . 42
3.2.2 A SQL-based Implementation of the Cox Model . . . 44
3.2.3 Initial Evaluation of the SQL-based Implementation of the Cox Model . . . 47
3.2.4 Distributed SQL-based Implementation of the Cox Model . . . 48
3.2.5 Integration with HDN and Future Challenges . . . 50
3.3 Public Datasets Supply for HEREDITARY project Needs . . . 50
3.3.1 LinkedLifeData Inventory . . . 51
3.3.2 Data Catalogue . . . 51
3.3.3 Data Loader . . . 52
3.3.4 Data Catalog Datasets . . . 54

4 Milestone Verification 55

5 Conclusions and Remarks 55

References 57

List of Tables

1 Iso-time at fixed endpoints: median supported scale and scale gain (HDN vs. OBDF). 30
2 Federation sensitivity: slowdown from 1 to 4 endpoints (E4/E1) at each scale (median latencies). 31
3 ALS clinical endpoints and local datasets used in the HDN workload. Patient counts refer to ALS patients or ALS–focused registries. 35
4 Use case L2 (average ALS age at onset) – local and federated results computed over the clinical datasets. 37
5 Why SQL-based algorithms for HDN? A summary of benefits. 41

List of Figures

1 Conceptual map covering the 4 main requirements areas that HDN addresses: (1) Regulatory Requirements (GDPR, DGA, AI Act); (2) Technical Requirements (heterogeneity, scalability, multimodality); (3) Institutional Requirements (privacy, autonomy, auditability); and (4) User Requirements (semantic queries, transparency, responsiveness). 13
2 Comparison between the D3.1 OBDF architecture (left) and the HDN model (right). Both are built on a stable ontology-mediated interface (HERO), but D3.1 centralizes query rewriting and relies on middleware without privacy-by-design, whereas HDN distributes semantic translation and integrates disclosure checks directly at each endpoint. 14
3 *Unified Modeling Language (UML)* interaction sequence diagram of the HDN hub-and-spoke topology: the user sends a parameterized template request to HDN Central, which (without accessing raw data) validates it and dispatches instantiated SPARQL queries, alongside template hash, to multiple HDN Endpoints. Each endpoint, acting as a local data custodian, translates the semantic query to its own storage, its disclosure policy, and returns result bindings or an empty answer (covering errors, authorization failures, and privacy refusals), which Central aggregates and sends back to the user. 18
4 HDN Central logical architecture. The figure showcases the interfaces with users and the network: users interact with a catalog of vetted queries; a dispatcher sends requests to all known participants and collects their answers. The aggregator combines the result sets, attaches provenance, and returns a consolidated relation. 20
5 *Variant Call Format (VCF)* to Relational transformation pipeline. The figure represents the pipeline of transformations and ingestion procedures a VCF file has to go through. VCFs after being uploaded and detected are transformed in relations by means of a dedicated module; next, they are ingested into an embedded database. 23
6 Experimental setup: data preparation, scaling, and deployment 26
7 OBDF baseline: per-query mean latencies (logarithmic) for each (*endpoints*, *scale*) panel. Error bars show variability. A compact core of fast queries coexists with a substantial set that grows with both data scale and involved endpoints. 28

8 HDN: per–query mean latencies (logarithmic) for each (*endpoints*, *scale*) panel. Error bars indicate variability. The trend mirrors the baseline but with a milder overall increase and a visibly shorter tail, especially at higher endpoint counts and larger scales. 29
9 Iso-time contour comparison (median latency surfaces). Left: HDN. Right: OBDF. Labels in contours show latency targets (seconds). Contours verticality in HDN indicates weaker dependence on endpoints; curves are also right-shifted, meaning larger scales are feasible at the same time target. 30
10 Improvement factor (OBDF/HDN) over the (*endpoints*, *scale*) grid (higher is better). Numbers are medians over queries for each cell. 31
11 HDN Central & Endpoints reference implementation. The figure showcases all the components and arrows connections outlines interactions among them. From bottom-up we can observe, for endpoints, the Data Ingestion module; the semantic layer represented by the ontology; the OBDA layer, exploiting the semantics and local mapping definition; the privacy tier, assessing incoming queries compliance with local settings. HDN Central consists of the orchestration of the query catalog, the query dispatcher, the results aggregator and the web *User Interface (UI)* for interaction with users. 32
12 End-to-end HDN cyclic workflow for the L2 *Amyotrophical Lateral Sclerosis (ALS)* onset-age query at the endpoint level. Starting from a catalog template, HDN Central dispatches the SPARQL query to all eligible endpoints, each of which rewrites it to *Structured Query Language (SQL)* through Ontop, executes it over local clinical tables, and returns a scalar aggregate. HDN Central merges the partial results into a federated answer, together with provenance and timing metadata. 37
13 ALS use case — per–query mean latencies (logarithmic) at fixed scale and increasing federation size (1 to 5 endpoints). Error bars show standard deviation. Most templates remain sub–second across the grid; a small set forms the long tail as endpoints increase. 39
14 Latency distributions (log–y) across federation sizes. Boxes summarize per–query latencies; dots identifies individual queries latencies. 40
15 Example of a SQL-based algorithm. 41
16 Schema of a survival dataset. In the example, each record has a *Patient ID* and five other attributes: the final recorded *Event*, the *Time* at which the *Event* occurred, and three covariates X_1, X_2 and X_3 42
17 Benchmark: comparison between our implementation and the faithful one. Our implementation asymptotically outperforms the naive one. 47
18 Comparison between our implementation and the state-of-the-art (scikit-learn). Our implementation is up to two order of magnitude faster than the state-of-the-art. 48
19 Architecture of a SQL-based implementation of the Cox model algorithm in a distributed environment. A host program issues queries to a SQL endpoint, receives the results, and post-processes them. Compared to Figure 15, the single database engine has been replaced by a DSE. Under the hood, the DSE issues queries to two different database engines, and collects the results. 49
20 Searching for datasets in the data catalog 52
21 Data Loader Dashboard of the subscribed datasets from Linked Life Data Inventory 53
22 Creating and Configuration of Data Catalog from LinkedLife Data Inventory 53
23 Configuring time schedule for a dataset updates 54

List of Acronyms

ALS Amyotrophical Lateral Sclerosis
BTO BrainTeaser Ontology
API Application Program Interface
ARP Advanced Relational Pushdown
BSBM Berlin SPARQL Benchmark
CPU Central Processing Unit
CSV Comma Separated Values
DBMS DataBase Management System
DCAT Data Catalog Vocabulary
DGA Data Governance Act
DPIA Data Protection Impact Assessments
DSE Distributed SQL Engine
ETL Extract-Transform-Load
FAIR Findable, Accessible, Interoperable, and Reusable
FVC Forced Vital Capacity
GDBMS Graph Database Management System
GDPR General Data Protection Regulation
HDN Hereditary Data Network
HEREDITARY HetERogeneous sEmantic Data integratIon for the guT-brAin inteRplaY
HERO Hereditary Ontology
HTTP HyperText Transfer Protocol
JSON JavaScript Object Notation
KGE Knowledge Graph Embeddings
KG Knowledge Graph
LLDI LinkedLifeData Inventory
LUBM Lehigh University Benchmark
NPD Norwegian Petroleum Directorate
OBDA Ontology-Based Data Access
OBDF Ontology-Based Data Federation
OLAP On-Line Analytical Processing
OWL Ontology Web Language
R2RML RDB-to-RDF Mapping Language
RDBMS Relational DBMS
RDF Resource Description Framework
SPARQL SPARQL Protocol And Query Language
SQL Structured Query Language
UI User Interface
UML Unified Modeling Language
UMLS Unified Medical Language System
URL Uniform Resource Locator
VCF Variant Call Format
VDS Virtual Data Set
VIG Virtual Instances Generator
VoID Vocabulary of Interlinked Data

1 Introduction

One of the primary objectives of the *HetERogeneous sEmantic Data integratIon for the guT-brAin inteRplaY (HEREDITARY)* project is to empower partners to perform advanced data analysis and model training across the consortium, strictly adhering to privacy and data re-use regulations. In the medical domain, sharing non-anonymized data is effectively prohibited by a complex regulatory landscape – including the *General Data Protection Regulation (GDPR)*, the Data Governance Act, and the emerging AI Act – which mandates that sensitive personal information remain under the control of its original custodians. This creates a critical barrier for multi-centric studies that traditionally rely on data centralization. To address this, this document introduces the HDN, a novel architecture designed not only to enable privacy-preserving query execution but also to overcome the inherent scalability and privacy limitations of traditional *Ontology-Based Data Access (OBDA)* approaches in federated settings. By shifting semantic translation and disclosure control from a central middleware to the edge nodes,HDN enables robust federated analytics while ensuring that data owners retain full autonomy. This work is presented within the context of Task 3.2, *Federated analytics and learning methods* of the HEREDITARY Project.

This deliverable builds upon the foundational work of the HEREDITARY project, specifically complementing the strategies for privacy-preserving machine learning introduced in D2.11 [24], *Federated infrastructure design*. That earlier work established an architecture based on the Flower Framework [5] to enable federated learning – training models locally on distributed nodes and aggregating updates on a central server without sharing raw data. While effective for training neural networks and other models on homogeneous data (e.g., medical imaging), such approaches typically require consistent schemas and code distribution across all participants. Addressing the complex reality of multi-modal and heterogeneous medical data, the present document focuses instead on federated analytics. We introduce a complementary model designed to execute semantic queries over diverse data structures, overcoming the rigidity of pure model-training frameworks. This architecture not only facilitates immediate data analysis but also establishes the necessary infrastructure for the future seamless integration of federated learning and federated analytics.

The core contribution of this deliverable is the introduction of federated execution methods tailored to heterogeneous, multi-modal medical data. Specifically, we present the HDN, an architecture that provides a unified semantic view of consortium data through the HERO ontology. By abstracting diverse local schemas into a single virtual data model, HDN enables researchers to formulate queries in *SPARQL Protocol And Query Language (SPARQL)* without requiring knowledge of institutional data structures. The architecture operates through a hub-and-spoke model: a central orchestrator validates query templates, enforces disclosure-level constraints, and dispatches SPARQL queries to participating endpoints. Each institutional node independently translates the ontology-level query into its local schema using OBDA mappings, executes it under local privacy policies, and returns admissible results – or remains silent if the request exceeds its disclosure threshold. The orchestrator then aggregates partial answers and provides provenance metadata, ensuring transparency while preserving institutional autonomy. This design realizes privacy-by-design at the protocol level and lays the groundwork for the future integration of federated learning with federated analytics.

1.1 Contributions

This document presents the first version of the HDN. In this context, this document presents:

- **HDN federation engine:** The HDN components, focusing on the central and local nodes with a description of each of the components that help the HDN provide services, focusing on architectural designs, including the interface, query compilation and dispatch, results aggregation, and privacy preserving elements.

- **Current implementation:** detailing the present state of the HDN and the concrete realization of its components, while also identifying current limitations and outlining improvements planned for future versions.
- **Experiment:** demonstrating the advantages of the current version of the HDN compared to other solutions, including the original version of the network.
- **Use cases:** providing specific scenarios that illustrate the advantages of the HDN. In particular, we provide use-cases on query data about ALS, learn Cox models through queries executed directly by *DataBase Management Systems (DBMSs)*, and retrieve data utilizing Ontotext's linked data portal.

This document is organized as follows. Section 2 presents the HDN, by discussing: (1) the evolution of the architecture from centralized middleware to a federated architecture, (2) the design principles and components of the HDN, including the communication protocols, the query execution mechanisms, the query life-cycle, and the privacy preserving components, and (3) an evaluation of the HDN, based on a benchmark consisting of 31 data analytics tasks, comparing it with OBDF alternatives. Next, Section 3 showcases three use cases for data analytics: (1) the use of HDN for ALS data analysis, with several queries that work with data in different levels of privacy, (2) how query engines can be used to fit a Cox survival model on centralised or distributed data, showcasing how HDN may support machine learning algorithms in future, and (3) the integration of HDN with Ontotext's *LinkedLifeData Inventory (LLDI)*, a *Findable, Accessible, Interoperable, and Reusable (FAIR)*-compliant data catalog (also related to Task 3.6 and D3.6). Section 4 discusses how this deliverable verifies Milestone 8: "federated workflow execution methods." Finally, Section 5 concludes the deliverable by discussing the current state of the HDN and its future uses.

2 The Hereditary Data Network

HDN is designed to address federated medical data analytics challenges, where clinically relevant information is distributed across multiple institutions, technologies, and regulatory regimes. At a high level, HDN tackles four distinct objectives: enabling semantic queries across heterogeneous and independently evolving data sources; enforcing privacy constraints through explicit disclosure levels; integrating multimodal data spanning clinical, genomic and imaging domains; and preserving institutional control over data by avoiding any centralization of sensitive records.

These challenges do not arise in isolation. Figure 1 describes how they are shaped by an articulated regulatory environment: regulations such as the GDPR, the *Data Governance Act (DGA)*, and the emerging AI Act impose strict constraints on how personal and health-related data can be accessed, processed, and reused. At the same time, large-scale medical studies require practical mechanisms to ask consistent questions across sites, compare patient cohorts, and build models without forcing institutions to abandon their existing infrastructures or give up ownership of their data. HDN is designed precisely at this intersection: it aims to provide a stable ontology-mediated interface to researchers, while allowing each institution to retain autonomy over storage, policies, and disclosure thresholds.

From a design perspective, HDN adopts a clear approach. Semantic access to data is mediated through a shared ontology (HERO [9]), allowing users to interact with a uniform conceptual view rather than with local schemas. Privacy is treated as a first-class concern: disclosure levels are defined at the semantic boundary of the system and enforced as part of the federation logic. Federation itself is organized so that endpoints, not the central node, remain the primary owners of data and are responsible for translating semantic queries into local operations, applying local policies, and returning only admissible results.

This section presents HDN along three complementary dimensions. Firstly, it introduces the principles that guided the evolution from the initial HEREDITARY architecture to the current HDN design, with particular

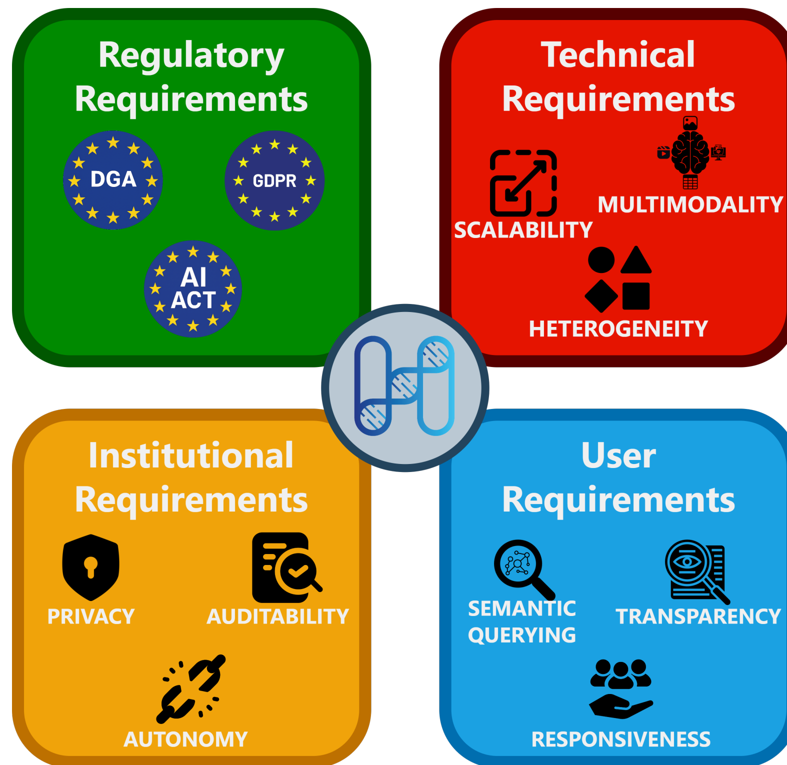


Figure 1: Conceptual map covering the 4 main requirements areas that HDN addresses: (1) Regulatory Requirements (GDPR, DGA, AI Act); (2) Technical Requirements (heterogeneity, scalability, multimodality); (3) Institutional Requirements (privacy, autonomy, auditability); and (4) User Requirements (semantic queries, transparency, responsiveness).

emphasis on semantic stability, privacy by design, and conservative execution. Secondly, it discusses the HDN architecture, describing the roles of the central orchestrator and the endpoints, together with their interaction protocol. Thirdly, it outlines the reference implementations of these components and benchmarks their performances, comparing them against the legacy HEREDITARY Federated Analytics architecture.

2.1 Architectural Evolution: From Centralized Middleware to Native Federation

The initial HEREDITARY architecture, introduced in D3.1 [10] and depicted in Figure 2 (left), provides a working solution for federated analytics across heterogeneous medical data sources. From a technological viewpoint, it instantiates an OBDF stack [15]: an OBDA layer exposes a unified conceptual schema in terms of the HERO ontology, while a data federation layer presents the union of the participating institutional databases as a single virtual relational source. In an OBDF system, users formulate queries at the ontology level (in our case, SPARQL over HERO); these queries are rewritten into executable SQL statements, distributed across the underlying sources, and evaluated as if the data were stored in one large database. This architecture supports cross-site joins and global analytics while preserving local autonomy: each site keeps its own technology stack and physical schema, yet contributes to a coherent semantic view. Moreover, OBDF naturally accommodates heterogeneity (relational warehouses, data lakes, analytical engines), leverages local query

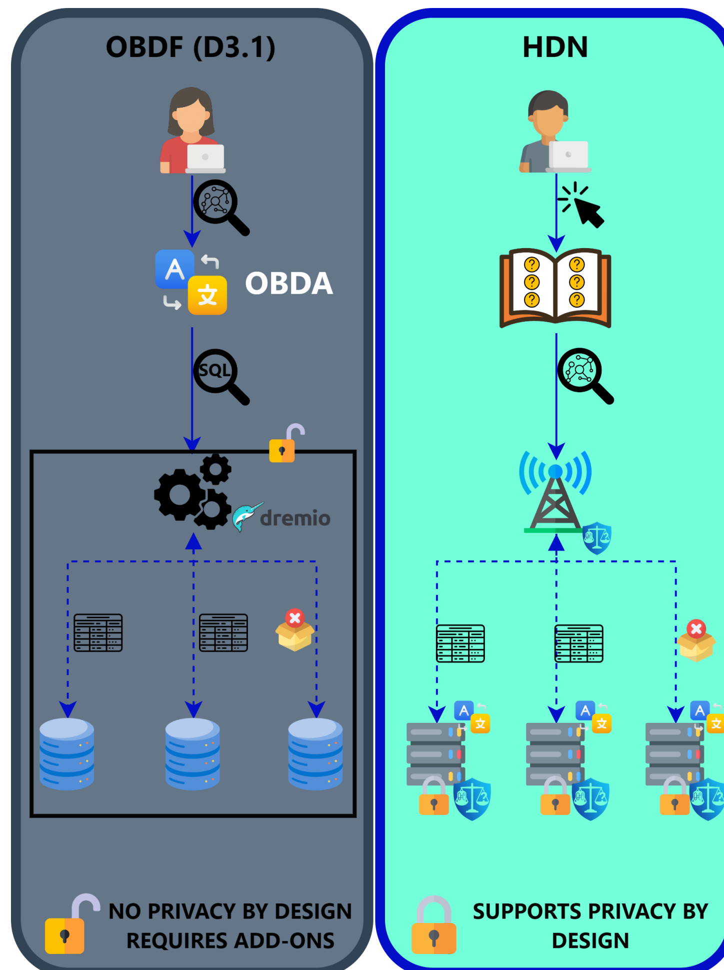


Figure 2: Comparison between the D3.1 OBDF architecture (left) and the HDN model (right). Both are built on a stable ontology-mediated interface (HERO), but D3.1 centralizes query rewriting and relies on middleware without privacy-by-design, whereas HDN distributes semantic translation and integrates disclosure checks directly at each endpoint.

optimizers and indexes, and avoids preventive bulk replication by relying on virtual integration at query time.

HDN, shown in Figure 2 (right), preserves the same ontology-mediated interaction model, but revisits how this stability is maintained in the presence of independently evolving local schemas and institutional constraints. Rather than concentrating semantic translation and orchestration in a central component, HDN distributes these responsibilities to the endpoints themselves. Each participating institution operates an endpoint that understands the shared ontology, maintains its own mappings to local structures, and takes responsibility for translating semantic queries into local operations. The central component coordinates template queries and aggregates results, but no longer needs detailed knowledge of the internal organization of each site.

This shift from a centralized middleware to local federation has important consequences for robustness and evolution. In the D3.1 architecture, changes in local schemas or storage technologies typically require updates at the central level, because the core mapping logic and federation behaviour are defined there. In

HDN, local network participants can adapt their internal schemas and pipelines as needed, as long as they keep their mappings consistent with HERO. Semantic stability is therefore achieved not by fixing a single central translation layer, but by enforcing a common conceptual contract at the boundaries of each endpoint.

Moving from centralized middleware to local federation has major implications for robustness and evolution, particularly for the federation's ability to continue delivering correct, predictable answers under realistic sources of instability, like (i) independent evolution of local schemas and storage technologies, (ii) partial failures such as temporary endpoint unavailability, timeouts, or degraded performance, and (iii) heterogeneous institutional policies that may change over time. In the D3.1 architecture, changes in local schemas or storage technologies typically require updates at the central level, because the core mapping logic and federation behaviour are defined there; similarly, the central translation layer becomes a single point whose misalignment can affect the whole network. In HDN, instead, each participant absorbs local evolution by updating its own mappings and execution pipeline while preserving the HERO contract, and disruptions remain localized: a failing, slow, or non-admissible endpoint only results in missing contributions from that site rather than breaking global execution. Semantic stability is therefore achieved not by fixing a single central translation layer, but by enforcing a common conceptual contract at the boundaries of each endpoint, which reduces coupling and limits the blast radius of local changes.

2.1.1 Principle: Privacy by Design

A key dimension of this evolution concerns privacy. In the D3.1 architecture, privacy constraints had to be layered on top of an execution stack that was not originally conceived for fine-grained disclosure control. The federation relies on a central orchestrator and an intermediate layer optimized for data access and integration; therefore, privacy logic must be introduced as additional checks and policies around these components, rather than as part of a uniform federation protocol. As a result, enforcing disclosure limits could depend on implementation-specific mechanisms and require careful coordination across systems that do not share a common notion of disclosure level.

HDN takes a different stance and treats privacy as a first-class architectural requirement. Disclosure levels are defined at the semantic boundary of the system, alongside the ontology, and are embedded in the notion of admissible query. Each request carries an explicit disclosure level, and each endpoint is required to evaluate this level against its local policy before touching the underlying data. If a query requests a disclosure level that exceeds the maximum allowed by a site, the endpoint does not attempt to execute it; instead, it returns an empty result that is indistinguishable from the absence of matching data. In this way, privacy enforcement is not an external add-on to federation, but an integral part of the protocol that governs how queries are interpreted and answered. In the following we present a use-case to further clarify the differences between the two architectures.

Example 1. *Consider a query that requests individual patient records with associated genetic variants at the highest disclosure level. Under the D3.1 architecture, the query would be translated at the central server and dispatched to all participating sites; ensuring that such a request respects local disclosure policies would typically require ad hoc checks external to the main federation logic. In HDN, the same query is expressed as a template ontology-level request annotated with a high disclosure level. Each endpoint independently compares this level with its configured maximum; sites that do not permit such detailed disclosure immediately return an empty result without executing the query, while others may execute it and return only admissible bindings. From the perspective of the central orchestrator, non-admissible sites are simply observed as returning no data, and their internal privacy configurations remain opaque.*

By embedding disclosure levels and privacy checks into the semantic and protocol layers, HDN aligns the federation mechanism with privacy-by-design principles. Control over what can be disclosed stays at the edge,

where institutions can define and revise their policies, while the central component remains agnostic to local rules and focuses on coordinating queries and aggregating whatever results are legitimately produced. This architecture thus reconciles a shared semantic interface for researchers with the need for strict, institution-specific control over sensitive medical data.

2.2 HDN Architecture: Components and Interactions

HDN is organized as a federation with a clear separation of roles between a central orchestrator and a set of institutional endpoints. The architecture follows a hub-and-spoke pattern: a single component, HDN Central, coordinates the execution of vetted query templates, while multiple nodes operated by participating institutions, HDN Endpoints, expose their local data through a shared semantic interface. This section describes these components at an architectural level, and explains how they interact during a typical federated query execution.

2.2.1 Architectural Components

HDN Central acts as the federation orchestrator. It curates a catalog of query templates, each defined over the HERO ontology and annotated with its admissible disclosure levels. Users do not submit arbitrary free-form queries, but work with parametrized prepared statements. The user can select a parametrized query template and provide the required parameters (based on their requirements) for the execution. HDN Central validates that the chosen templates are compatible with the global network catalog. It then instantiates the templates into a concrete semantic query, records the request for auditing purposes, and dispatches them to the available endpoints. Crucially, HDN Central never stores or accesses raw patient-level data; its role is limited to coordinating execution, collecting answers and provenance, and aggregating or combining result sets.

Each HDN Endpoint is operated by a data curator, such as hospitals, research centers, or external registry. Endpoints expose their local data through the shared ontology, translating ontology-level concepts into queries over local schemas and storage technologies. They are responsible for enforcing local disclosure policies, implementing the mapping between HERO and their internal structures, and executing the concrete queries derived from HDN templates. From the perspective of HDN Central, endpoints are black boxes that accept semantic queries annotated with disclosure levels and either return admissible bindings or remain silent; all details on how data are stored, ingested, and processed stay under institutional control.

This separation of responsibilities reflects the design goals introduced in the previous section. HDN Central provides an entry point for users over the data, and it offers a common coordination mechanism to manage the query distribution and final answer computation; HDN Endpoints retain autonomy over their data and policies. As long as endpoints uphold the ontology contract and the disclosure semantics, they are free to evolve their internal schemas, add new modalities, or reconfigure their infrastructures without requiring changes to the central orchestrator.

2.2.2 Privacy Levels

Data privacy in HDN is managed through a set of discrete privacy levels that are defined at the semantic boundary of the system and attached to each query template in the shared catalog.[1] Each template is annotated with a level that encodes the maximum type and granularity of information that may be disclosed. HDN Endpoints interpret these levels against their local policies before touching any underlying data: if a request

[1] See Annex 1 for the full list of clinical templates and their assigned levels.

exceeds the locally permitted level, the endpoint returns no result, making privacy refusals indistinguishable from the absence of matching data. This mechanism allows HDN to support a wide range of analytical needs while keeping sensitive medical data under the control of institutional custodians.

The HDN query catalog currently distinguishes six main privacy levels, ordered from least to most permissive in terms of data granularity:

- **L1 – Boolean / count queries:** Only global existence checks or aggregate counts, without exposure of patient-level characteristics.
- **L2 – Statistical aggregations:** Summary statistics such as means, medians, or survival times over broad groups, without returning individual records.
- **L3 – Grouped aggregations:** Stratified or multi-dimensional aggregations (e.g., by age bracket, onset type), subject to minimum group-size constraints (e.g., k-anonymity).
- **L4 – Attribute selections (no direct identifiers):** Cohort-level selections returning non-identifying attributes (e.g., age, sex, diagnosis, onset characteristics) but excluding direct identifiers.
- **L5 – Anonymised records / complex analytics:** Row-level data with pseudonymous identifiers suitable for statistical modelling, where direct identifiers are removed and re-identification risk is explicitly assessed.
- **L6 – Identifiable records / full access:** Full patient profiles including direct identifiers and event histories, reserved for strictly controlled clinical or operational scenarios.

The detailed catalog of clinical query templates and their mapping to these privacy levels is provided in Annex 1, which illustrates for each level concrete SPARQL examples and parameterisations over HERO.

2.2.3 Interaction Protocol

Communication between HDN Central and HDN Endpoints follows the standard semantic query protocol adopted in HEREDITARY. At a high level, each request sent by Central to an Endpoint consists of the identifier of the selected template encoded as a cryptographic hash; the instantiated semantic query obtained by filling the template with user-provided parameters.

Since the catalog is shared between HDN Central and HDN Endpoints, once local validation succeeds, HDN Endpoints can locally check the query privacy level without the need to expose this information in the request payload, enforcing network resilience against external manipulations. Each response contains either a set of result bindings or a boolean answer in the case of existence checks, together with minimal provenance, such as the endpoint identifier and a timestamp reference. Transport errors, authorization failures, and explicit privacy refusals are all normalized to the same observable behavior: the endpoint does not contribute to the result set, thereby preserving privacy-by-default semantics at the protocol level.

Figure 3 shows the interaction sequence among users and both HDN Central and endpoints. This interaction is conceived as a short-lived and stateless exchange. HDN Central does not open long-running sessions with endpoints; instead, each federated query leads to a bounded burst of requests that can be processed in parallel. Endpoints are free to apply their own scheduling and optimization strategies, but from the network's point of view they either answer within a configured time window or are treated as non-responding for that query. This design simplifies failure handling and avoids coupling the success of the federation to the behavior of any single site.

Example 2. *Consider a researcher interested in the number of patients diagnosed with ALS who exhibit a specific biomarker, at a low disclosure level that only allows for aggregated counts. The researcher selects*

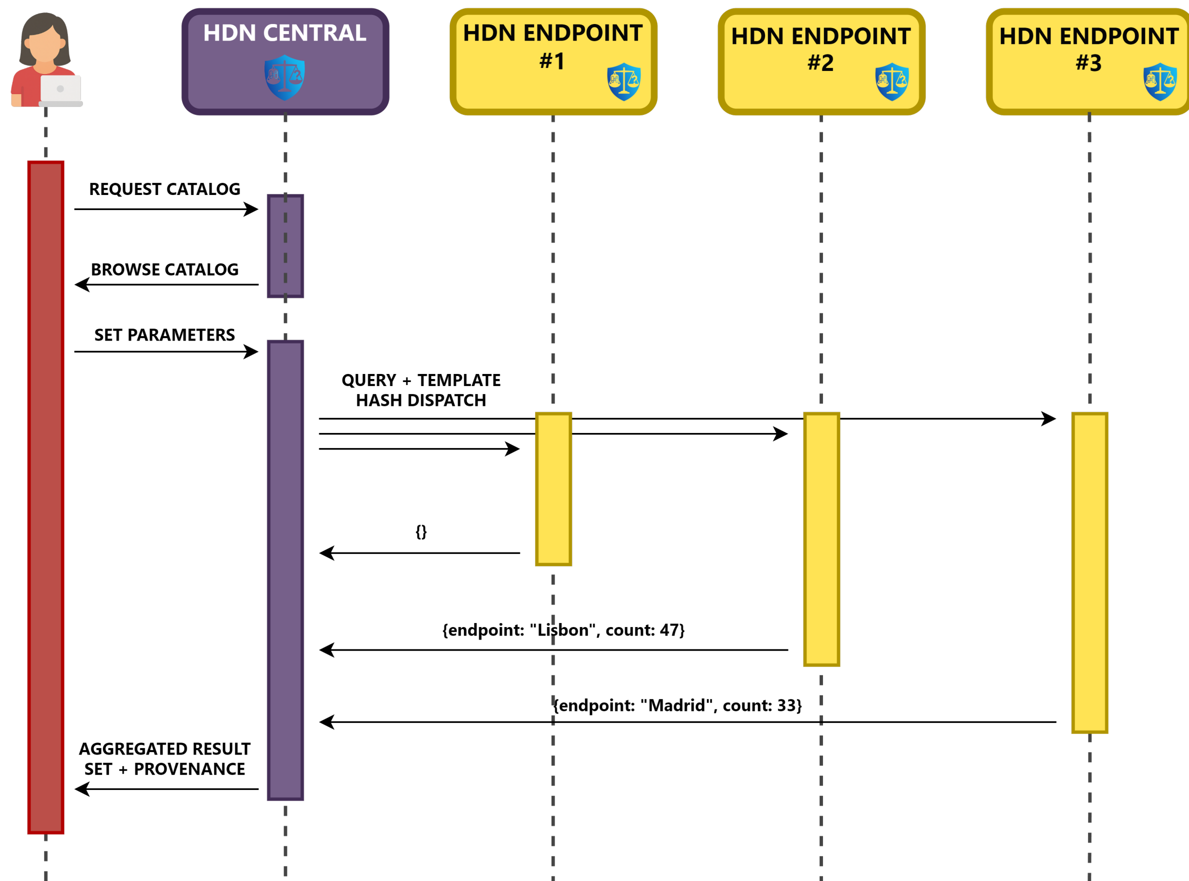


Figure 3: UML interaction sequence diagram of the HDN hub-and-spoke topology: the user sends a parameterized template request to HDN Central, which (without accessing raw data) validates it and dispatches instantiated SPARQL queries, alongside template hash, to multiple HDN Endpoints. Each endpoint, acting as a local data custodian, translates the semantic query to its own storage, its disclosure policy, and returns result bindings or an empty answer (covering errors, authorization failures, and privacy refusals), which Central aggregates and sends back to the user.

the appropriate template from the catalog (annotated with level L1) and provides the biomarker identifier as a parameter. HDN Central verifies that the template is consistent with the local catalog; then, it instantiates the template into a concrete semantic query and broadcast it to five registered endpoints, corresponding to different centers in Lisbon, Turin, Madrid, Padua, and a large synthetic cohort. Each endpoint independently checks that L1 does not exceed its locally configured maximum, unfolds the semantic query to its local schema, executes the aggregation, and either returns a binding such as {`endpoint: "Lisbon", count: 47`} or returns no bindings at all if the request is not admissible or if no matching patients exist. HDN Central collects these partial answers and aggregates them into a global result, for instance reporting a total count of 161 patients considering as contributions 47, 32, 18, 0, and 64, respectively, across the five sites. From the researcher's perspective, the final answer provides also a provenance summary indicating which endpoints have contributed.

2.3 HDN Central: the HDN Federation Engine

This section introduces HDN Central, the federation engine of HDN. HDN Central is the sole orchestrator of query execution across the network: it curates the privacy-based catalog of semantically meaningful queries, compiles template requests into executable queries, dispatches them to the HDN Endpoints, and computes the final result set by combining the partial answers it receives.

2.3.1 Design Principles

HDN Central is designed to achieve the objectives of HEREDITARY Task 3.2: a multi-modal semantic polystore system that enables federated querying and federated analytics; compliance with heterogeneous institutional constraints, unified access to clinical, omics, and image-derived data, and a substrate for AI-assisted tasks. These objectives are achieved at query time without centralizing partners' data.

Two design commitments follow from this premise: (1) semantics are stable at the interface: users interact with an ontology-aligned view, independent of the storage choices of partners, and (2) execution is conservative: the engine minimizes data movement, treats "no answer" as a valid outcome, and records the provenance of any result returned.

Conservative execution means that HDN Central does not assume that all endpoints will always respond, nor that they will respond within a fixed time. Instead, it operates under a best-effort contract: it dispatches a query to the registered endpoints, waits up to a configured timeout, and proceeds with whatever answers arrive within that window, treating non-responses and explicit refusals as admissible outcomes that must be made visible in provenance.

Example 3. *Consider a network-wide query that is dispatched to five endpoints, one of which is temporarily unreachable due to a network partition. Rather than retrying indefinitely or failing the entire federation, HDN Central aggregates the answers received from the four responding sites and returns them to the user, accompanied by coverage metadata indicating that four out of five endpoints contributed. The non-respondent is visible at the provenance level but does not prevent the study from progressing.*

These principles align the behavior of HDN Central with the requirements discussed in the previous sections: a stable semantic interface over heterogeneous data nodes, explicit disclosure control, and a conservative, provenance-aware approach to federation.

2.3.2 Logical Architecture of HDN Central

At a logical level, HDN Central is organized into four cooperating layers that form a pipeline from user interaction to federated execution and back to consolidated results. Figure 4 summarizes these four layers and their interactions, highlighting how user-level operations are translated into federated requests and how the results are combined into a single provenance-aware response.

At the top, the *Interface Layer* exposes the network's query catalog and the interaction surface for users and client applications. It presents templates (covertly grouped by disclosure level) alongside their description, generates parameter forms from the template schemas, and renders either unified or aggregated answers, with per-site provenance. The same interface can surface analytics-oriented interactions, such as inspecting distributions or simple models derived from aggregated outcomes.

Beneath the interface, the *Catalog and Policy Layer* manages query templates as curated artifacts defined on the HDN ontology. Each template carries a unique identifier (a SHA-512 hash of the template text), a natural-language description, a list of parameters with their types, and a disclosure level indicating the admissible granularity of answers (existence checks, targeted aggregates, or record-level results). HDN Central

uses this layer to certify the templates identity for outgoing requests and to grant the endpoints the ability to execute concrete SPARQL queries.

The third layer, the *Instantiation and Dispatch Layer*, is responsible for constructing executable requests and sending them to the endpoints. It prefixes ontology namespaces, instantiates parameters in the template body, and forms a request that includes both the instantiated SPARQL query and the original template identifier. For dispatch, HDN Central contacts partner endpoints through a protected web interface dedicated to template execution. Dispatch is designed to be resilient: timeouts are applied; transport or authorization failures are treated as non-fatal; and HDN Central proceeds with processing any partial answers that arrive within the window. No assumptions about the internal behavior of endpoints beyond the contract are made that a request yields either an ASK result, a set of SELECT bindings, or an empty set.

Finally, the *Aggregation and Reporting Layer* collects returned bindings and normalizes them into a single relation where each row retains its responder of origin; ASK and record-level results are presented per responder, while global aggregations are computed, with sites contribution indication. This layer also reconciles datatypes and label encodings and deduplicates by variable keys, where applicable. It also produces the final views exposed to users, ranging from sortable tables to summary charts. For analytics templates, HDN Central can compute central-side metrics (for instance, distributional comparisons) or train simple statistical models against the aggregate, returning fit metrics and visual artifacts together with a model identifier when applicable.

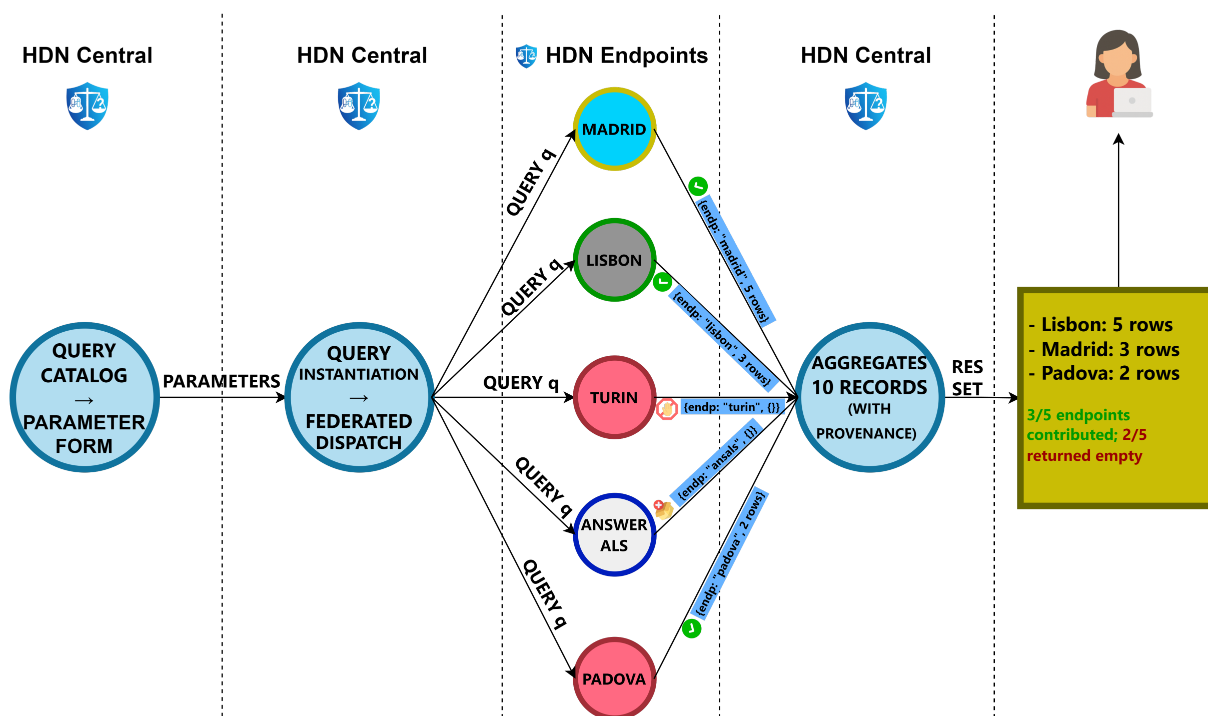


Figure 4: HDN Central logical architecture. The figure showcases the interfaces with users and the network: users interact with a catalog of vetted queries; a dispatcher sends requests to all known participants and collects their answers. The aggregator combines the result sets, attaches provenance, and returns a consolidated relation.

2.3.3 Federated Query Execution

A request begins with the user selecting a template query from the catalog and providing the input values for its parameters. HDN Central validates the instance against the template's schema and constructs the SPARQL query by concatenating the ontology prefixes with the template body and substituting placeholders. The instantiated query and the hashed original template (acting as its identifier) are transmitted together to participants. The original template hash is required because it enables remote verification of legitimate requests, without exposing a free-form interface.

Example 4. *Consider the logical query plan in Figure 4 as a privacy-sensitive study that requires patient-level genetic data at the highest disclosure level (L5). A researcher selects template* `T_GeneticVariants_L5` *from the catalog, providing the gene symbol* `C9orf72` *as parameter. HDN Central validates, instantiates the corresponding SPARQL query, hashes the template and dispatches this packet to five endpoints within a timeout window. The endpoints respond as follows: Lisbon returns three matching records, Turin returns an empty result because its local policy does not permit L5 disclosure, Madrid returns five records, Padua returns two records, and an external cohort returns an empty result because no matching patients are present. HDN Central aggregates the ten records contributed by Lisbon, Madrid, and Padua, and returns them to the researcher together with a note that 3/5 endpoints contributed data while 2/5 returned empty results, indistinguishable between data absence and privacy refusal.*

During execution, the engine awaits replies up to the configured timeout and collects three kinds of outcomes: boolean values for ASK, bindings for SELECT, and empty results when a responder has no data to disclose at the requested level, declines to answer, or fails to respond in time. DESCRIBE and CONSTRUCT queries are not supported at the moment, but they will be introduced if required by the use cases. Emptiness is recorded explicitly and does not count as an error: HDN Central never assumes that all endpoints will answer; instead, its provenance model records which responders contributed to a result and which were silent or returned emptiness. Once the replies arrive, HDN Central reconciles the data types and label encodings, deduplicates them using variable keys where applicable, and computes any final aggregations required by the template. The answer is delivered alongside responder provenance and execution metadata. Users can therefore also interpret answers in terms of coverage statistics such as "3/5 endpoints contributed", and downstream workflows can decide how to react to partial federation, for instance by repeating a query at a lower disclosure level (where this still meets their needs) or by combining results with external knowledge.

A minimal registry records each responder with a display name, an *Uniform Resource Locator (URL)*, and an optional logo used in provenance views; only authenticated managers can add or edit network participants. Alongside this registry, the query catalog stores each template together with a stable hash identifier, the disclosure level, the template body, a parameter schema and a description that helps users to understand what they are querying for. The hash preserves catalog adherence across endpoints, the disclosure reflects the granularity of answers, and the parameter schema and descriptions serves for form generation in the user interface.

2.4 HDN Endpoints: the Local HDN Nodes

This section presents the design principles of the HDN Endpoint, the local query engines deployed at each participating institution to expose local data to the HDN under a shared semantic vocabulary. Moreover, the section presents a reference implementation of the HDN Endpoints based on an OBDA layer. It is worth noting that further implementations of HDN Endpoints are possible.

2.4.1 Role and Design Principles

The HDN Endpoint represents the interface between an institution's proprietary data and the network. It accepts catalogued graph queries from HDN Central and answers them against native data without any prior centralization. To do so reliably, it maintains a semantic alignment with HERO, it ensures its execution (e.g., by directly executing it over a SPARQL engine or by translating it into SQL queries over the institution's *Relational DBMS (RDBMS)*), and returns bindings or an explicit empty answer when data is unavailable or disclosure is not allowed at the requested privacy level.

Two principles guide the design of the HDN Endpoint. First, the external contract is semantic and stable: users always query an ontology-aligned view, while local schemas and storage technologies are free to evolve as long as they remain backward compatible with HERO; in practice, new concepts and relations can be introduced, whereas schema changes and deletions must be remapped to the ontology. Second, execution is conservative and verifiable: HDN Endpoints maintain a local copy of the HDN Central query catalog, enabling disclosure compatibility checks to be performed at the endpoint without exposing disclosure metadata in each request, while filters and projections are pushed down and each answer is accompanied by complete local provenance. As a concrete consequence, HDN Central can compute coverage indicators and derive informed assessments about data availability across the federation.

However, the endpoint's functionalities necessarily extend beyond query answering. It offers controlled facilities to ingest and register local assets, author and validate mappings, configure disclosure policies, and observe health and performance. These functions are present to the extent necessary to keep semantics and operations coherent across sites, to fully support data models heterogeneity and to ease use experience to clinicians and analysts who rely on the network for decision support and research.

2.4.2 Logical Architecture

From an architectural point of view, an HDN Endpoint is organized into a small number of logical layers that structure how requests are received, checked, interpreted, rewritten and executed. At the boundary, a service layer authenticates incoming requests from HDN Central and validates that they correspond to known templates. Immediately behind it, the privacy layer evaluates the requested disclosure level against the locally configured maximum and decides whether a query is admissible before any access to the underlying data occurs. Once a request is admitted, a semantic layer hosts the relevant HERO modules to execute the semantic query (e.g., through OBDA). Finally, the HDN Endpoint prepare a response with the resulting bindings, together with provenance information.

This logical flow keeps concerns separate: authentication and template verification at the edge, privacy decisions before data access, semantic interpretation over the ontology, mechanical rewriting and unfolding to local structures, and execution over native storage.

2.4.3 Architecture and Data Pipeline of the Reference HDN Endpoint

We propose a reference rechitecture HDN Endpoint based on the OBDA paradigm, as we think it will be the most common implementation required by the HEREDITARY use-case partners. A front-end interface provides essential management functions to local administrators: a service layer authenticates and validates requests from HDN Central; the semantic tier holds the relevant HERO modules (HERO-Clinical and HERO-Genomics) together with *RDB-to-RDF Mapping Language (R2RML)*[2] mappings that bind ontology terms to a normalized relational view. The input semantic patterns are rewritten with respect to the ontology axioms and

[2] https://www.w3.org/TR/r2rml/

unfolded into SQL over the local view before execution by the embedded engine. Results are standardized for datatypes and encodings and returned with minimal provenance.

The HDN Endpoint realizes an OBDA stack by embedding an Ontop [8] instance. R2RML mappings declare how HERO classes and properties are populated from the local relational view. The query rewriting and answering workflow is then carried out in three steps: (1) semantic rewriting expands the incoming pattern with respect to HERO axioms (e.g., subclass/sub-property inclusions, value-set alignments) and normalizes it into a union of conjunctive queries over ontology predicates, (2) unfolding replaces each ontology predicate with its mapping SQL, composing filters and projections so that constants, ranges, and joins are pushed down as far as possible, and (3) the unfolded query is executed in the embedded *On-Line Analytical Processing (OLAP)* engine (e.g., DuckDB [23]). The endpoint's service layer authenticates and validates template based requests from HDN Central, the semantic tier hosts HERO-Clinical and HERO-Genomics alongside active mappings, and the analytical engine provides a single SQL dialect with high performance. results are shaped in a standard format according to the definition of the SELECT clause of the query and returned with minimal provenance information that identifies the endpoint.

This architecture yields source independence (schemas can evolve under the condition of maintaining consistent mappings), cross-modal integration (clinical and genomics through shared identifiers in HERO), and auditability without centralizing data.

This reference implementation assumes the existence of an OLAP engine that stores the data and can answer queries. Therefore, if the institution holds data, for example, in *Comma Separated Values (CSV)* files, administrations should upload such files and register them as local tables within the OLAP engine. Basic schema inference is performed at load time; mapping templates are then human-instantiated from the endpoint manager by binding HERO predicates/classes to real columns.

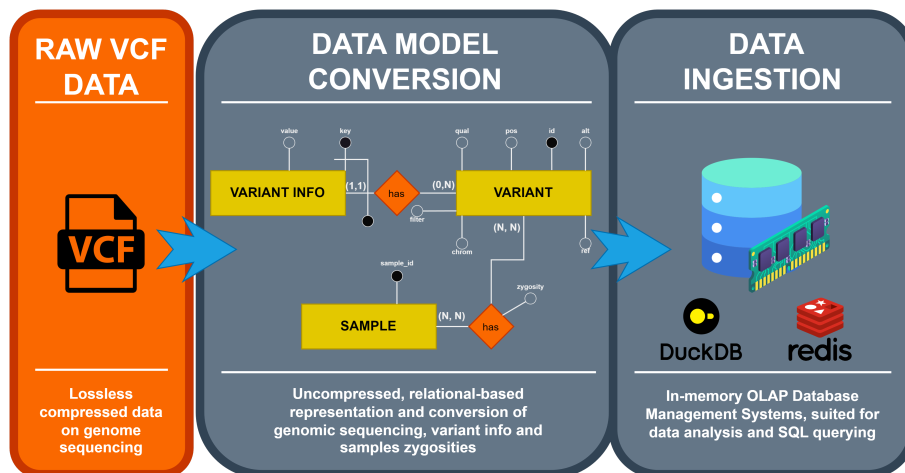


Figure 5: VCF to Relational transformation pipeline. The figure represents the pipeline of transformations and ingestion procedures a VCF file has to go through. VCFs after being uploaded and detected are transformed in relations by means of a dedicated module; next, they are ingested into an embedded database.

For genomics data, the reference HDN endpoint includes a VCF pipeline, presented in Figure 5, that preserves informative content while enabling efficient and ontology-aligned querying. It parses headers, body, and metadata; normalizes core columns into a variant table (decomposing multi-allelic sites), flattens INFO attributes into a typed key–value structure and expands FORMAT/sample data into a genotype table. Stable identifiers are assigned to the variants and the sample and links to the original record are retained for trace-

ability. Ontology mappings to HERO-Genomics allow predicates such as variant type, locus ranges, allele matching, and genotype filters to unfold into SQL patterns, and support cross-modal joins with clinical data in the same semantic domain.

Example 5. *A research center may contribute genomic sequencing data in VCF format comprising roughly 2.1 GB, 450,000 variants, and 50 samples. The endpoint parses the file to extract sample identifiers, decomposes multi-allelic sites into normalized variant rows, flattens informative attributes such as allele frequency and predicted consequence into typed columns, and ingests the resulting tables into the embedded analytical engine with indexes on chromosome, position, and gene symbol. Through HERO-Genomics mappings, a semantic query such as "patients with `C9orf72` variants" is unfolded into a compact SQL statement over this representation, for instance selecting patient identifiers from the variant table where the gene symbol equals `C9orf72`.*

2.4.4 Querying and Privacy Control

Disclosure is governed at the service boundary. Hence, the only HDN Endpoints responsibility is to configure the highest level they are eager to support (e.g., L0 existence checks, L1 targeted aggregates, L2 cohort counts with brackets, L3 record level retrieval for specific variables, etc.). This approach concatenates with a template-based acceptance policy: non-catalogued queries are discarded even before the semantic translation process.

If the requested level is not admissible or if the query template is not present in the local catalog, the endpoint immediately returns an empty result set and records the event locally for audit. From the perspective of HDN Central, such responses are indistinguishable from sites where no matching data exist.

Example 6. *Consider a query that requests individual patient ages and treatment responses at disclosure level L4, while the local policy only allows grouped aggregations up to level L3. Rather than executing the query and filtering results after the fact, the endpoint compares L4 with its configured maximum, determines that the request is not admissible, and returns an empty result without touching the underlying tables. The refusal is logged internally, but what the federation observes is merely the absence of bindings, on par with a site where no patients match the requested criteria.*

2.5 Comparison of the Previous Architecture with the New Architecture

This section compares the D3.1 architecture with the HDN architecture on the *Norwegian Petroleum Directorate (NPD)* workload through a bidimensional analysis, varying both the number of participating endpoints and the dataset scale. We retain the canonical NPD query set and report per-query and aggregated latencies. This analysis is showcased through three complementary views: (i) bar plots that expose per-query execution time; (ii) a bidimensional improvement heatmap (baseline/HDN) that summarizes median speedups; and (iii) iso-time contours over median surfaces that make scaling behaviour visually explicit.

The following subsections detail the experimental setup. The NPD benchmark and its faithful adapters for OBDF and for HDN will be described in details; subsequently, results for each of the two implementations will be presented, followed by a comparison through the improvement heatmap and the iso-time contours. We conclude with a query-level inspection to identify which patterns benefit most from HDN execution.

2.5.1 Experimental Benchmark

When evaluating the performance of a DBMS (such as RDBMS), it is common practice to adopt well-known benchmarks built on synthetic datasets. Typically, data are generated on purpose to debias the experiments

from specific dataset instances; subsequently, datasets are scaled at different sizes to observe the systems' evolution behavior. Well-known queries, designed to stress different DBMS constructs (e.g., indexes, primary/foreign keys, etc.) are executed and latencies are recorded. Given our scenario and the novelty of our approach, we explored existing benchmarks for OBDA-like systems that would have required the lowest effort for federated adaptation. We identified three existing benchmarks: the *Berlin SPARQL Benchmark (BSBM)* [6], the *Lehigh University Benchmark (LUBM)* [16] and the NPD benchmark [19]. Among these three, we chose to adopt the NPD benchmark to evaluate our system's performance, as it was the only one specifically designed for OBDA systems (LUBM was originally designed for RDBMS, a porting has been realized recently [2]; BSBM was originally intended for *Graph Database Management Systems (GDBMSs)*, there exist an unofficial adaptation[3] for OBDA systems that comes without guarantees).

The NPD Benchmark[4] is an OBDA oriented suite built from the Norwegian Petroleum Directorate FactPages and designed to exploit the full SPARQL-SQL rewriting pipeline, under an ontology-mediated view. It packages five assets: (i) a relational instance derived from FactPages; (ii) an OWL-2 QL ontology; (iii) a SPARQL query set; (iv) R2RML mappings, and (v) tooling for scaling and automation (*Virtual Instances Generator (VIG)*). Together, these elements make NPD a faithful implementation of an real-world industrial OBDA workload, where query performance depends jointly on ontology reasoning, mapping structure, and relational execution. FactPages are materialized into a relational schema with 70 tables, 276 distinct attributes (about one thousand columns overall), and 94 foreign keys; redundancy across tables is intentional and mirrors the source organization. The seed instance is modest but is explicitly meant to be scaled by VIG.

The benchmark adopts the *Ontology Web Language (OWL)* 2 QL fragment of the NPD ontology, yielding 343 classes, 142 object properties, 238 data properties and 1451 axioms. OWL 2 QL guarantees first-order rewritability of unions of conjunctive queries, making the suite fitting for testing reasoning over hierarchies and existentials without leaving the SQL execution regime. The benchmark curates 31 SPARQL queries that span filters, optional patterns, aggregates, and reasoning-sensitive constructs. The mapping layer comprises 1190 R2RML assertions covering 464 ontology symbols. The set is intentionally non-minimal so that OBDA engines can expose their mapping-level optimization strategies at load time.

VIG [20] grows the underlying relational instance by a given factor while preserving statistics that matter for virtual (ontology-level) cardinalities and for physical constraints (primary and foreign keys, datatypes). Its analysis phase estimates duplication and value distributions; the generation phase injects duplicates and fresh values within learned ranges, keeping join sizes realistic for NPD's mostly key-equality joins.

2.5.2 Experimental Setup

The remodeling process for NPD involved several steps, from data preparation (conversion and scaling), to architecture configurations (both for OBDF and HDN), towards the actual experiments execution. The NPD reference repository[5] contains the original NPD dump, a generator synthesizing the original dump, the VIG scaler and 31 benchmark queries spanning different SPARQL constructs, such as filters, oredering, use blank nodes, aggregation strategies and bindings. All of these components provide support for MySQL and PostgreSQL as SQL dialects: given our current reference implementation, this required a dialect transformation procedure. Furthermore, we found the data generator providing non-compatible outputs with respect to modern MySQL and PostgreSQL versions. Moreover, we tried to use VIG on the original dumps to leave out only the parsing preparation step to be accomplished, but even this component was not running properly. Given these considerations, we decided to reimplement the whole data preparation pipeline.

[3] https://github.com/mchaloupka/bsbm-r2rml

[4] https://ontop-vkg.org/npd-benchmark/v1.10/local-index.html

[5] https://github.com/ontop/npd-benchmark

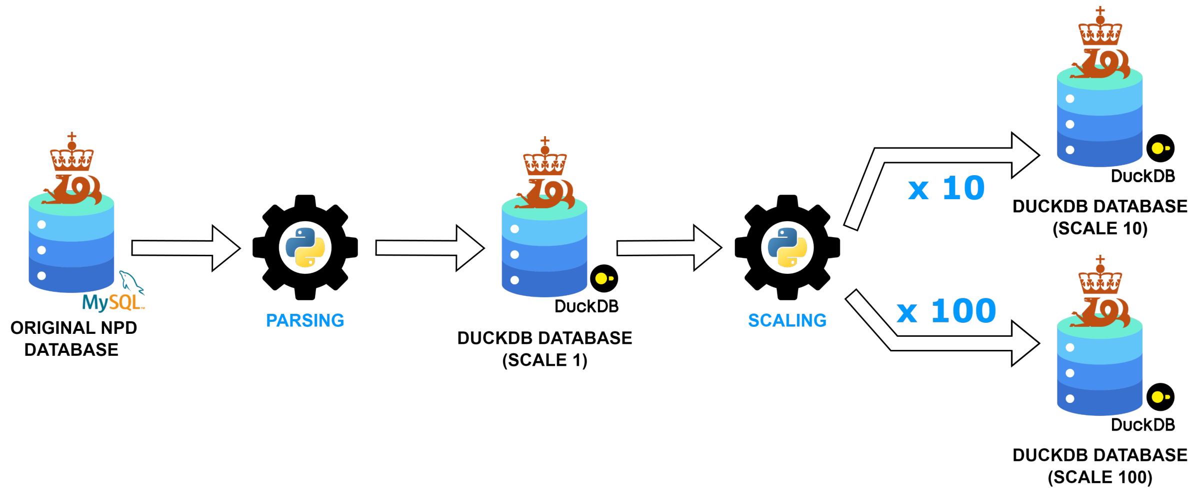


Figure 6: Data preparation pipeline for the NPD benchmark: starting from the official MySQL dump, a Python converter produces DuckDB-compatible SQL; a Python, DuckDB-like scaler generates $\times 10$ and $\times 100$ instances.

Figure 6 points out the steps we took: we considered the MySQL original dump available in the repository that consists in our scale 1 dataset. At first, we developed a Python converter, transforming the MySQL dialect into DuckDB compatible SQL directives, and stripping out unsupported operations. After obtaining a DuckDB compliant dataset, we developed a Python-based, DuckDB-like VIG scaler, with a conservative approach in terms of average rows and constraints per table. We ran the script on the DuckDB $\times 1$ dataset to obtain both a $\times 10$ and a $\times 100$ scaled datasets. Subsequently, we set up the OBDF baseline architecture and the HDN network, preparing them to host these datasets. These two processes were not straightforward. We considered our baseline OBDF implementation using Dremio[6] as the Data Federation component, which does not provide native support for DuckDB. Luckily, its inherently modular nature allowed contributors to develop a custom *Advanced Relational Pushdown (ARP)* driver[7]. At the time of our implementation it required some minimal code fixes to run properly due to relative paths bugs. We created then 12 different Dremio data sources (4 for each scale) and 12 different Dremio *Virtual Data Set (VDS)*, one for each experiment. VDS, for each scale, were defined as the UNION ALL of n sources, where n is the number of endpoints considered $(n \in \{1, 2, 3, 4\})$. An example of VDS definition as virtual view is reported in Annex 2.

HDN experimental architecture preparation required us to disable some aforementioned modules, such as catalog lookup and privacy mechanisms, as they would have hampered the experiment execution; NPD datasets, mappings and ontology have been replicated in four distinct endpoints. In this scenario, endpoint selection (1 to 4) and union for each of the three scales was set at the experiment level. We tested in both architectures all the 31 queries provided from the repository: we found two of them (Q12 and Q24) not running because of incompatibilities with the ontology, as Ontop was triggering exceptions; therefore, we excluded them from our experiment. For each query, we specified warm-up runs to ignore caching delays and multiple measured runs, considering the mean latency; moreover, we set three back-off and jitter dimensions or on-failure delay according to this logic:

$$D_k = \text{backoff_initial} \cdot (\text{backoff_multiplier})^k + X_k$$

[6] https://www.dremio.com/

[7] https://gitlab.inf.unibz.it/obdf/util/dremio-connector-duckdb

where $X_k \sim \mathcal{U}(0,\ \text{jitter})$ is a uniform random variable between 0 and the specified jitter value. In both of our experiments, we set the same parameters: specifically, 1 warm-up execution and 3 measured executions; initial back-off set to 1 second, back-off multiplier to 2 seconds and jitter equal to 0.3 seconds.

2.5.3 OBDF Baseline Results

Figure 7 reports per-query mean latencies (logarithmic) for the OBDF baseline across the 12 independent experiments, scaling both the size of the synthetic dataset (1, 10 and 100) and the number of endpoint involved (1 to 4). Each panel fixes a pair endpoint-scale within the grid. Bars show the average latency time for each of the 29 considered queries, with error bars indicating the standard deviation.

Three observations emerge. First, latency increases monotonically with scale at fixed endpoints; some queries remain compact while many others stretches drastically. This delta is already visible at between scales 1 and 10, and becomes even more evident if we consider the largest scale. Second, increasing the number of endpoints at fixed scale amplifies the average latency: in other words, federation increases timing costs even without data growth: by observing slowdown at fixed scales, we observe in our experiments a n average slowdown factor of $\approx 2.59\times$ (from 1 to 4 endpoints). Third, variability (the error bars) results modest for the fast half of the workload and larger for the extreme tail, consistent with small network/scheduling jitter interacting with heavy intermediates in the shared backend. Reading the grid left to right (increasing endpoints) and top to bottom (increasing scale) makes the pattern explicit: panels elongate vertically and a handful of templates dominate the timing budget, especially in the bottom right region.

2.5.4 HDN Results

Figure 8 reports per–query mean latencies (logarithmic) for HDN across the same 12 experiments used for the baseline, still varying dataset scale (1, 10, 100) and the number of endpoints (1–4). Again, as before each panel fixes an *endpoints–scale* pair; bars show the average latency of the 29 queries with error bars indicating the standard deviation.

Three observations mirror the baseline, but with visibly milder effects. First, latency still grows monotonically with scale at fixed endpoints, but the overall lift of each panel is less severe than in OBDF; the long tail is attenuated and a larger fraction of templates remains compact even at the largest scale. Second, increasing endpoints at fixed scale does raise latency, but more gently: across scales, the average slowdown from 1 to 4 endpoints is $\approx 1.67\times$, substantially below the baseline trend. Third, run–to–run variability stays modest for the fast core and is reduced in the tail compared to the baseline, consistent with less central materialization pressure.

Reading the grid left to right and top to bottom, panels elongate vertically but to a smaller extent than OBDF. These observations highlights how the two systems behave under scaling scenarios, both in terms of data size and federation. Furthermore, it gives a first glimpse of how better, in terms of federation, HDN scales with respect to OBDF. However, these two independent analysis fails to capture improvements among the two systems.

2.5.5 Comparative Analysis

We compare the OBDF baseline to HDN on the full $\{\text{scale} \in \{1, 10, 100\}\} \times \{\text{endpoints} \in \{1, 2, 3, 4\}\}$ grid using median per–query latencies to form two bidimensional surfaces. The iso-time contours in Figure 9 render equal latency curves over each surface, while the ratio heatmap in Figure 10 summarizes pointwise speedups as OBDF/HDN (higher is better), making scaling behavior explicit.

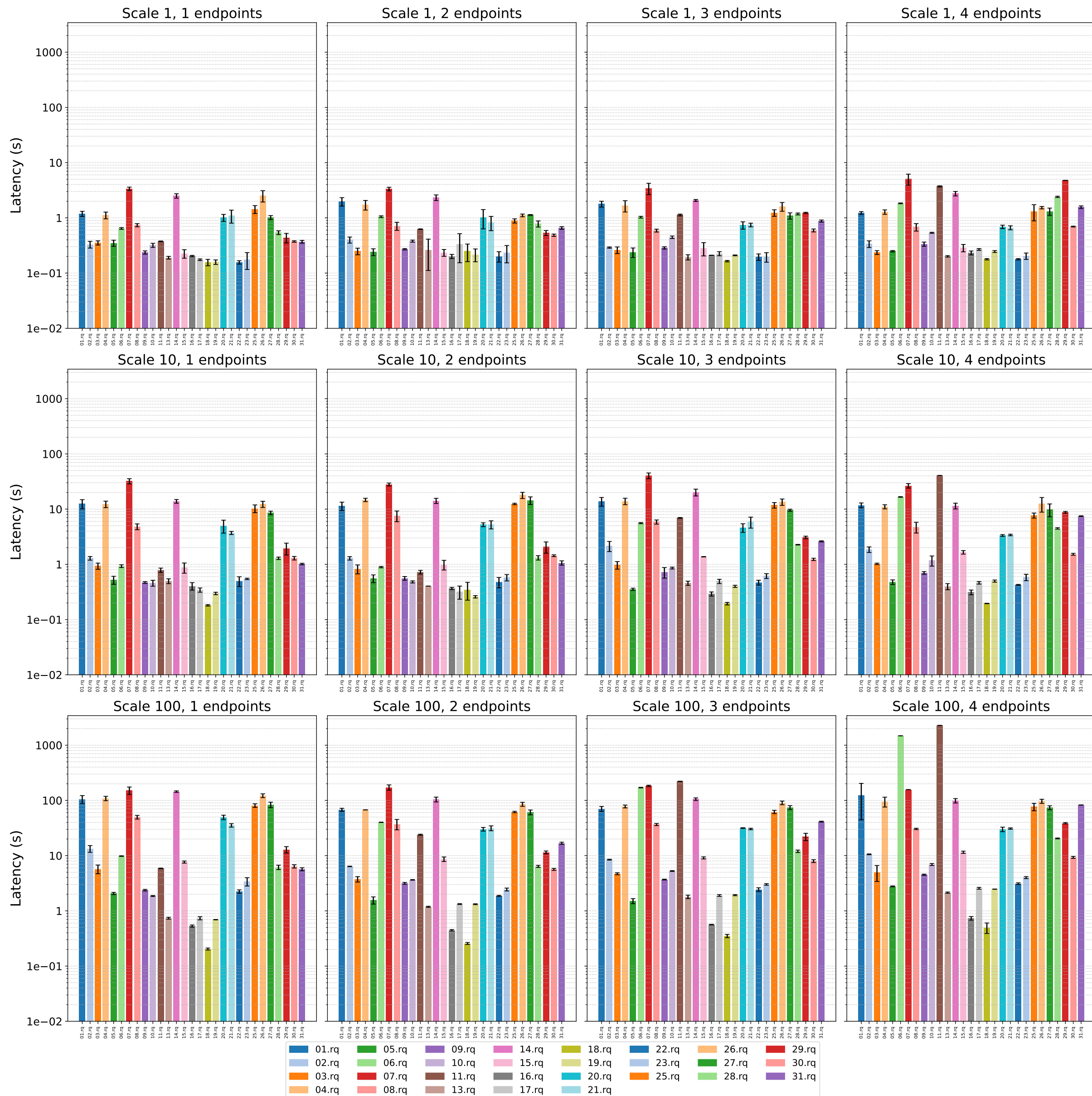


Figure 7: OBDF baseline: per-query mean latencies (logarithmic) for each $(\textit{endpoints}, \textit{scale})$ panel. Error bars show variability. A compact core of fast queries coexists with a substantial set that grows with both data scale and involved endpoints.

Across the heatmap grid in Figure 10, HDN is consistently faster than the OBDF baseline in all the configurations, with median improvements typically in the $1.5\times$–$4.3\times$ range (mean $2.78\times$). The largest speedup occurs at $\textit{endpoints} = 4,\ \textit{scale} = 1$ with $4.30\times$; the smallest at $\textit{endpoints} = 1,\ \textit{scale} = 100$ with $1.55\times$.

Federation growth (from single endpoint up to 4 endpoints) hurts OBDF more than HDN. Using mean

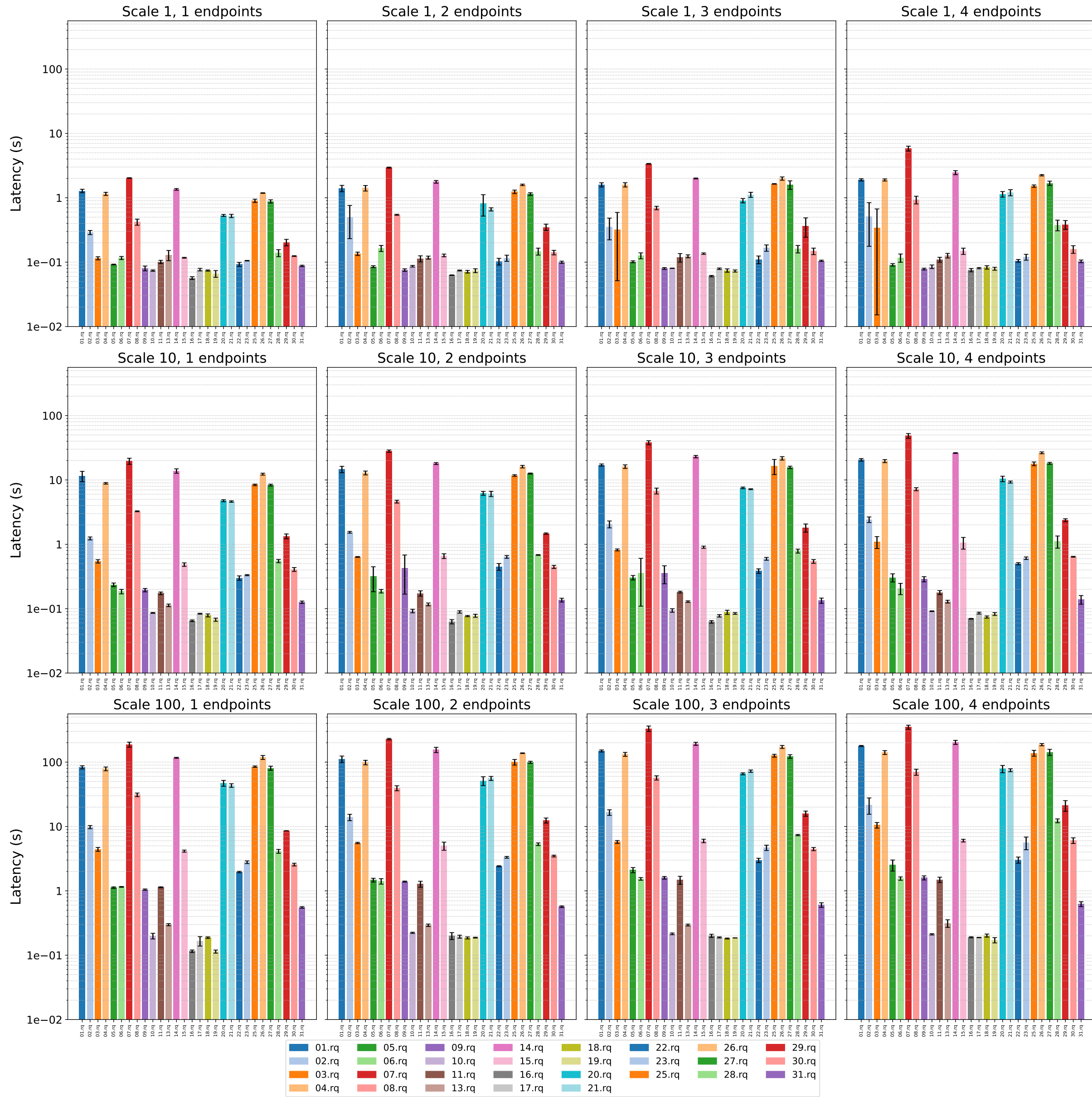


Figure 8: HDN: per–query mean latencies (logarithmic) for each (*endpoints*, *scale*) panel. Error bars indicate variability. The trend mirrors the baseline but with a milder overall increase and a visibly shorter tail, especially at higher endpoint counts and larger scales.

latencies (considering median timing for each experiment, to exclude outliers), the slowdown factor from 1 to 4 endpoints computed as:

$$\tilde{T}_{s,e} := \text{median}_{q \in Q_{s,e}} \, T_{s,e,q}, \quad (1)$$

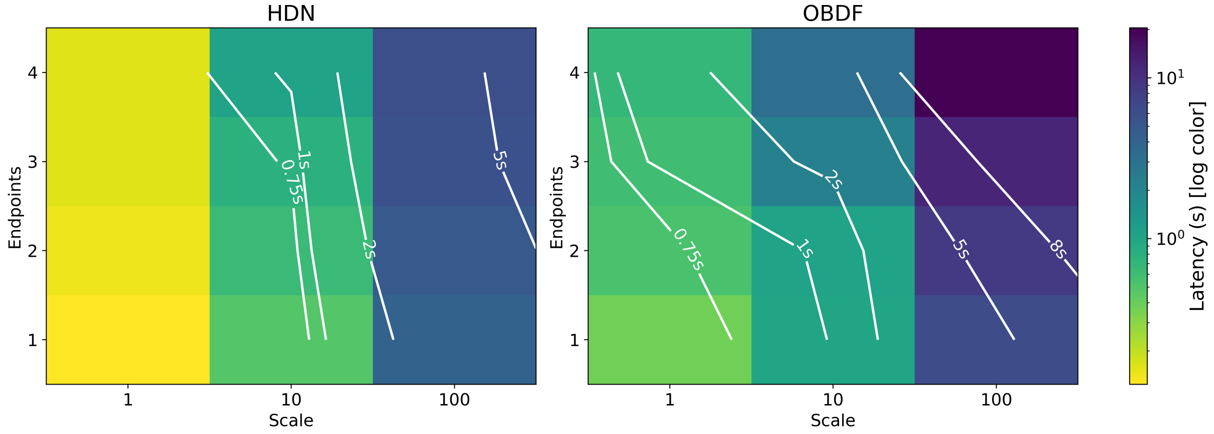


Figure 9: Iso-time contour comparison (median latency surfaces). Left: HDN. Right: OBDF. Labels in contours show latency targets (seconds). Contours verticality in HDN indicates weaker dependence on endpoints; curves are also right-shifted, meaning larger scales are feasible at the same time target.

$$S^\star := \{\, s \mid \tilde{T}_{s,E_{\min}} \text{ and } \tilde{T}_{s,E_{\max}} \text{ defined} \,\}, \qquad E^\star := \{\, e \mid \tilde{T}_{1,e} \text{ and } \tilde{T}_{100,e} \text{ defined} \,\}, \tag{2}$$

$$\mathrm{SF}_{\text{endp}} := \frac{1}{|S^\star|} \sum_{s\in S^\star} \frac{\tilde{T}_{s,E_{\max}}}{\tilde{T}_{s,E_{\min}}}, \tag{3}$$

$$\mathrm{SF}^{1\to 100}_{\text{scale}} := \frac{1}{|E^\star|} \sum_{e\in E^\star} \frac{\tilde{T}_{100,e}}{\tilde{T}_{1,e}}. \tag{4}$$

Averages $\times 1.64$ for OBDF vs. $\times 2.77$ for HDN across scales (see Table 2 for scale-based slowdown breakdowns).

Table 1: Iso-time at fixed endpoints: median supported scale and scale gain (HDN vs. OBDF).

Target (s)	OBDF	HDN	×Gain	Range	#E
0.50	2.78	10.38	3.74	3.74–3.74	1
1.00	4.64	15.69	3.38	2.52–4.92	4
2.00	5.34	14.80	2.77	1.84–3.29	4
5.00	17.00	57.52	3.38	2.92–3.83	2

In other words, HDN reduces the penalty induced by the endpoint by approximately **41%**. In contrast, multiplicative growth with data size (scale 100 vs. 1) averages $\times 20.91$ for OBDF and $\times 35.64$ for HDN. This larger factor for HDN reflects its much smaller baseline latency at scale 1 (e.g., with 1 endpoint: 0.12s vs. 0.37s), not worse absolute times at scale 100. In fact, at the hardest cell $(4, 100)$ we observe OBDF $= 20.49$s vs. HDN $= 5.84$s, a $\times 3.51$ gain.

Another important analysis concerns the temporal persistence of the performances in the two different configurations. Figure 9 highlights two stable patterns: first, HDN iso-time lines are *closer to vertical*, indicating weaker dependence on the number of endpoints; OBDF lines are in contrast more slanted, meaning that the same latency target requires fewer endpoints for the baseline as scale increases. Second, curves in the HDN panel are *systematically right-shifted* with respect to OBDF: this implies that, at the same latency target, HDN

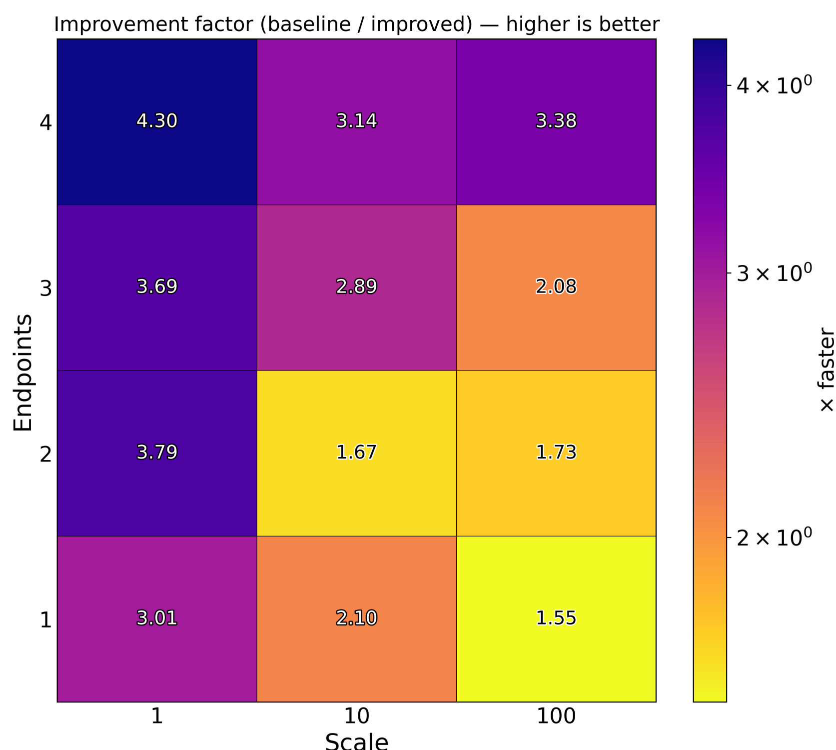


Figure 10: Improvement factor (OBDF/HDN) over the $(endpoints, scale)$ grid (higher is better). Numbers are medians over queries for each cell.

sustains substantially larger scales. Quantitatively, at fixed endpoints, the median scale supported by HDN at the same time target exceeds OBDF by roughly $\times 2.8$–$\times 3.7$ depending on the iso-level (Tab. 1); for example, at the 1 second target the median supported scale improves from ≈ 4.64 to ≈ 15.69 ($\times 3.38$).

Table 2: Federation sensitivity: slowdown from 1 to 4 endpoints (E4/E1) at each scale (median latencies).

Scale	OBDF	HDN	OBDF/HDN
1	1.82	1.27	1.43
10	3.28	2.19	1.50
100	3.20	1.47	2.18

Averages across scales: OBDF $\times 2.77$; HDN $\times 1.64$.

Taken together, these results show that HDN delivers broad, robust gains: (i) per-cell median speedups lie mostly between $\times 2$ and $\times 4$ with the largest advantages where federation costs dominate; (ii) endpoint-induced slowdowns are markedly lower under HDN; and (iii) at equal latency targets, HDN sustains on average 3 times larger scales (often more) with respect to OBDF. The bar plot panels in Sections 2.5.3 and 2.5.4 corroborate this picture: the OBDF tail grows sharply with endpoints and scale, while HDN remains more compact and stable across the grid.

2.6 Implementation Notes

The previous sections have described HDN Central and HDN Endpoints at the level of principles and logical architecture, without committing to specific technologies. This section briefly summarizes the current reference implementations developed within HEREDITARY. These implementations are meant to validate the design in realistic scenarios and to provide a working substrate for experimentation, but they are not prescriptive: logical choices promote rapid prototyping and may evolve as the project progresses. Reference code is made available as open-source software on GitHub, so that partners can inspect, reuse, or adapt it to their own environments.

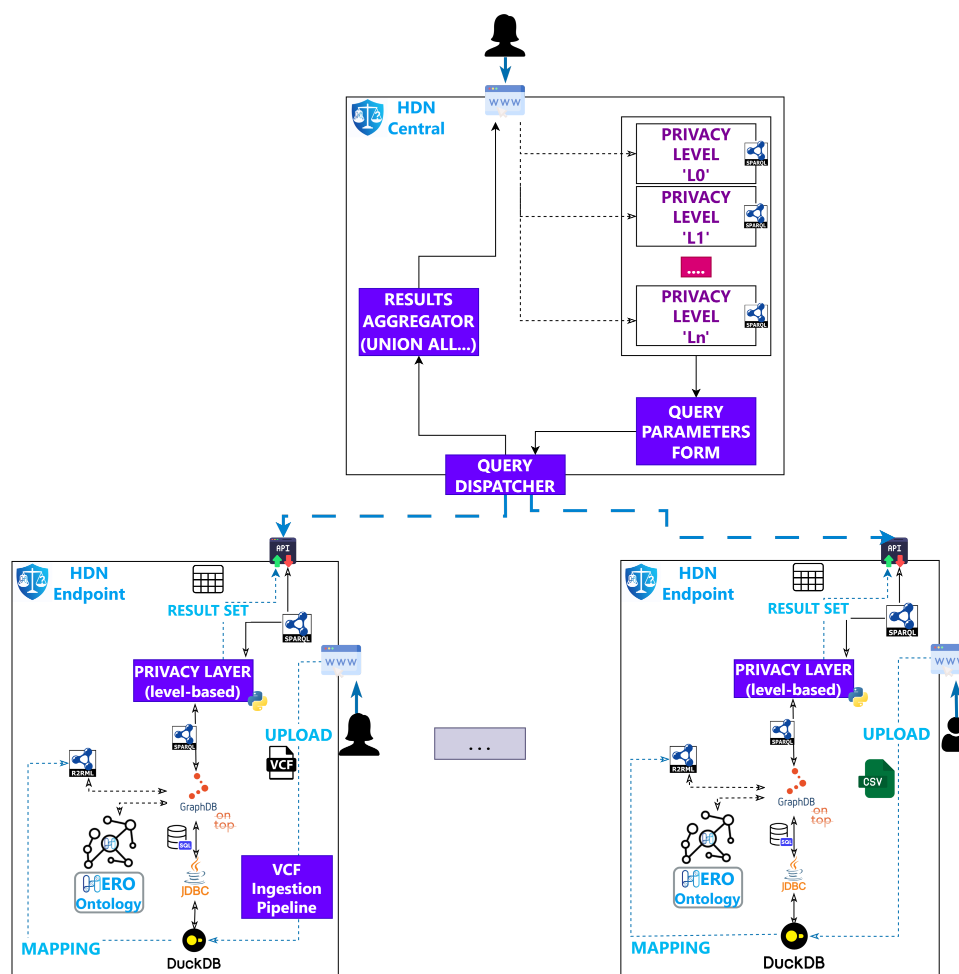


Figure 11: HDN Central & Endpoints reference implementation. The figure showcases all the components and arrows connections outlines interactions among them. From bottom-up we can observe, for endpoints, the Data Ingestion module; the semantic layer represented by the ontology; the OBDA layer, exploiting the semantics and local mapping definition; the privacy tier, assessing incoming queries compliance with local settings. HDN Central consists of the orchestration of the query catalog, the query dispatcher, the results aggregator and the web UI for interaction with users.

2.6.1 Reference Implementation of HDN Central

The reference implementation of HDN Central is built as a web application that exposes both a human-oriented interface and a programmatic *Application Program Interface (API)*. The service is delivered as a Django application; the user interface relies on Bootstrap components and a small amount of client-side scripting to implement catalog browsing, parameter form generation, interactive result tables and analytics visualization units. Consolidated answers are rendered as HTML tables that can be sorted and filtered, and, when relevant, are complemented by summary charts. Results can also be exported for offline analysis.

Internally, the query catalog is implemented as a Python module that defines trusted query templates as structured objects. Each template carries a stable hash identifier, a disclosure level, a parameter schema, and a textual description, mirroring the logical model described in Section 2.3. Parameter substitution is explicit and type-checked before any dispatch takes place. The dispatch layer uses *HyperText Transfer Protocol (HTTP)* POST to contact responders' protected SPARQL endpoints, attaching both the instantiated query and the template identifier. Responses are normalized, merged, and passed to the presentation layer together with provenance and execution metadata.

When analytics templates are enabled, HDN Central integrates lightweight scientific tools on the server side to compute metrics over aggregate results or to fit simple models such as regressors or classifiers. The corresponding artifacts (fit reports, basic plots, and model identifiers) are returned alongside the main query result, so that users can interpret patterns in the federated data without exporting them to external environments. Deployment requires only a standard Python web stack and an HTTP(S) front-end; no specialized infrastructure is assumed beyond the existence of reachable SPARQL endpoints at the participating sites.

2.6.2 Reference Implementation of HDN Endpoints

The reference HDN Endpoint is implemented according to the reference architecture described in Section 2.4.3. It is a modular web service that combines a semantic integration core with an embedded analytical engine and a simple administration interface. Local administrators interact with the endpoint through a web UI to upload data, inspect registered tables, manage mappings, and configure disclosure policies. Requests from HDN Central are served through a dedicated service interface that authenticates callers, verifies template identifiers, and enforces disclosure-level constraints at the boundary.

The semantic integration stack realizes an OBDA setting by embedding and orchestrating an Ontop instance. HDN Central issues SPARQL queries over HERO, while sources remain in their native schemas and formats. R2RML mappings specify how ontology classes and properties are populated from a normalized relational view. Query answering proceeds in three phases, as described in Section 2.4.3: semantic rewriting with respect to HERO axioms, unfolding to a union of SQL queries over the local view, and execution over an embedded OLAP engine, such as DuckDB. The endpoint shapes results into *JavaScript Object Notation (JSON)* according to the original SELECT clause and attaches minimal provenance identifying the responder.

Data ingestion is handled by dedicated components. For tabular sources, administrators upload CSV files, which are registered as tables within the embedded analytical engine. Basic schema inference is performed at load time, and optional constraints and indexes can be declared to improve push-down and query performance. Mapping templates are defined through a visual interface and instantiated by binding HERO predicates and classes to concrete columns in these tables.

For genomics data, the endpoint provides a VCF processing pipeline that preserves the informative content of variant calls while making them accessible through the ontology. The pipeline parses headers and records, normalizes core fields into a variant table (decomposing multi-allelic sites), flattens INFO attributes into typed columns, and expands FORMAT/sample information into a separate genotype table. Stable identifiers are assigned to variants and samples, and links to the originating VCF records are retained for traceabil-

ity. Through HERO-Genomics mappings, ontology-level predicates such as variant type, locus ranges, allele matching, and genotype patterns unfold into compact SQL queries over these tables, enabling cross-modal joins with clinical data at the semantic level.

The same endpoint service also implements the privacy and resource controls described earlier. Disclosure levels are configured as local settings that the service layer checks before a query is admitted to the semantic pipeline; non-admissible requests result in empty answers and internal log entries, but no access to the underlying data. Timeouts, per-template concurrency limits, and a maintenance mode provide basic protection for local resources and support gradual deployment across sites with heterogeneous infrastructures. Users can actually toggle the desired privacy level to which they aim to adhere at any time through a dedicated select input.

In general, these implementations demonstrate that the HDN architecture can be realized with standard web technologies, an existing OBDA engine, and a lightweight embedded analytical backend, while keeping institutions in control of their data, mappings, and disclosure policies. As the project evolves, alternative implementations may adopt different frameworks or storage systems, provided that they maintain the semantic, privacy, and protocol contracts specified in this chapter.

We prepared a video to illustrate the the HDN reference implementation workflow. [8] At the endpoint level, operators can perform actions described in Section 2.4.3: onboarding new data sources, one-time mapping operations, privacy-oriented configurations and testing locally semantic and structured queries. Globally, users can query the network, selecting queries from a dedicated catalog and parametrizing them.

3 Use Case Studies

This section presents three different case studies. Firstly, Section 3.1 discusses how HDN can be used for querying ALS data relative to the Use Cases 1 and 2 of the project. The section includes a discussion on the implications of privacy levels and the number of endpoints. Secondly, Section 3.2.1 illustrates how a machine learning model, the Cox survival model, can be implemented through a query engine. This sets the path towards the extension of HDN to machine learning workloads. Thirdly, Section 3.3 shows how Ontotext's linked data portal can be used for retrieving distributed data.

3.1 Query Distribution on ALS Data

We evaluated HDN on a real clinical workload built from ALS data mapped to HERO[9]. The federation spans five heterogeneous endpoints: two BRAINTEASER [13] nodes (Lisbon, Madrid), one HEREDITARY data source (Turin, revised data from the BRAINTEASER project), one Padua University Hospital node, and one Answer ALS [4] node (currently focusing on clinical metadata). Unlike the synthetic NPD study, here we fix the data scale (real-world data; no synthetic scaling) and vary only the federation size.

3.1.1 Local Data: Quantity and Characteristics

Table 3 summarizes the local datasets contributing to the clinical workload. Three of them (Lisbon, Madrid and Turin) expose the *BrainTeaser Ontology (BTO)* schema [12] with: (i) a *static* per-patient table (demographics, onset and diagnosis dates, comorbidities, genetics, etc.; 46 attributes including age_onset); (ii) a longitudinal ALSFRS/ALSFRS-R table with repeated functional scores per visit; and (iii) a spirometry table with repeated

[8] https://hereditary.dei.unipd.it/demo/demo_tdn.mp4
[9] https://hereditary.dei.unipd.it/ontology/

Forced Vital Capacity (FVC) measurements. The Padua node contributes an independent clinical registry with an anagraphic table (ID, birth date, diagnosis date, etc.), a detailed anamnesis table (104 variables, including onset age `etEsordio`) and a visit table with neurological follow-ups. The Answer ALS endpoint exposes a curated subset of the public portal, with about one thousand participants and a small ALS subcohort; in this study we only use the harmonized clinical metadata mapped to HERO (diagnosis category, sex, participation metadata).

Table 3: ALS clinical endpoints and local datasets used in the HDN workload. Patient counts refer to ALS patients or ALS–focused registries.

Endpoint	#Patients	#ALSFRS rows	#Spirometry / visits
Lisbon (BRAINTEASER)	1307	6902	2333
Madrid (BRAINTEASER)	173	1212	361
Turin (HEREDITARY/BRAINTEASER)	1853	14397	2513
Padua University Hospital (HEREDITARY)	50	–	256
Answer ALS (clinical subset)	1041	3379	–

Overall, the clinical ALS workload covers more than $3{,}300$ ALS patients across Europe, with over $22{,}000$ ALSFRS visits and about $5{,}200$ spirometry records from the BRAINTEASER/HEREDITARY nodes, plus a smaller but semantically rich clinical subset from Answer ALS. All local schemas are mapped to HERO, so that patients, events and clinical measurements are exposed through a unified ontology-level view.

3.1.2 Query Templates and Privacy Levels

On top of HERO, we defined 15 query templates (q0–q14) spanning privacy levels L0–L6, from boolean existence checks (L0) up to full patient profiles with identifiers (L6). Templates are expressed in SPARQL over HERO and are parameterized along a small number of axes (disease concept, age thresholds, dates, sex, etc.). They cover: simple counts and boolean lookups (L0–L1), aggregate statistics such as averages and distributions (L2–L3), limited access to non–sensitive data (L4), anonymized patient-level views (L5), and fully identified clinical traces (L6). In the experiments below, the disease parameter is instantiated to ALS, represented as `NCIT:C34373` in the ontology.

The listing 1 exemplifies a low-privacy template (L0), which answers the boolean existence query “Is there any patient diagnosed with `<DISEASE>`?”:

```
PREFIX bto: <https://w3id.org/brainteaser/ontology/schema/>
PREFIX : <https://w3id.org/hereditary/ontology/phenoclinical/schema/>
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

ASK WHERE {
  ?pat a bto:Patient ;
       bto:hasDisease NCIT:C34373 .
}
```

Listing 1: Example L0 boolean template (q0): existence of ALS patients.

The query returns true as soon as at least one local endpoint exposes a patient diagnosed with ALS. Higher levels build on the same vocabulary, but expose progressively more informative outputs (aggregates, distributions, anonymized lines, full profiles). The full list of templates is provided in Annex 1.

3.1.3 Use case: Federated Average Age at Onset for ALS (L2)

To make the clinical workload tangible, we now focus on a single L2 template presented in Listing 2, that we use as running example in the experiments: *"What is the average age at onset of ALS patients?"*. At the ontology level, the query computes the mean of bto:ageOnset across all ALS patients in the federation:

```
PREFIX bto: <https://w3id.org/brainteaser/ontology/schema/>
PREFIX : <https://w3id.org/hereditary/ontology/phenoclinical/schema/>
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

SELECT (AVG(?aO) AS ?avgAge) WHERE {
  ?pat a bto:Patient ;
       bto:hasDisease NCIT:C34373 ;
       bto:undergo ?eO .
  ?eO a bto:Onset ;
       bto:ageOnset ?aO .
}
```

Listing 2: Use-case L2 template (q2): average ALS age at onset.

At the HDN catalog level, this appears as a single template with a fixed disease parameter (NCIT:C34373) and annotated for disclosure as belonging to level L2 (aggregated, non-identifying output). The corresponding end-to-end workflow is summarized in Figure 12: the clinician selects the template from the HDN Central catalog, the federated engine instantiates it, dispatches it to all endpoints that declare ALS coverage, and collects a single scalar average in response.

Input, mapping and execution. Figure 12 decomposes the L2 use case into four stages: *Input*, *Mapping*, *Execution* and *Output & timings*. At the *Input* stage, the user issues the high-level query "average ALS age at onset" through the web UI; HDN Central resolves it to the catalog entry in Listing 2.

At the *Mapping* stage, each endpoint uses its local R2RML mapping to expose the relevant clinical tables as HERO individuals. For instance, at the Turin node, the static registry table `Turin_static_vars` contains one row per ALS patient, with a numeric `age_onset` field. Similarly, Lisbon and Madrid employ analogous mappings over their local `Lisbon_static_vars` and `Madrid_static_vars` tables, where `age_onset` is represented with the same semantics. The Padua endpoint instead maps onset age from the anamnesis field `etEsordio` in `als_sampledata_anamnesi`, exposing it again as `bto:ageOnset`. Answer ALS contributes with ALS cases where onset age is available in the harmonized metadata and mapped to HERO. Mapping definitions snippets per endpoint are reported in Annex 3.

In the *Execution* stage, HDN Central sends the instantiated template to all endpoints. Each endpoint uses Ontop to rewrite Listing 2 into an SQL aggregate over its local tables (e.g., a `SELECT AVG(age_onset) ...` query over DuckDB or PostgreSQL). Local results thus consist of a single scalar average per endpoint; HDN Central returns to the user a two columns table, indicating endpoint provenance information and the average onset value.

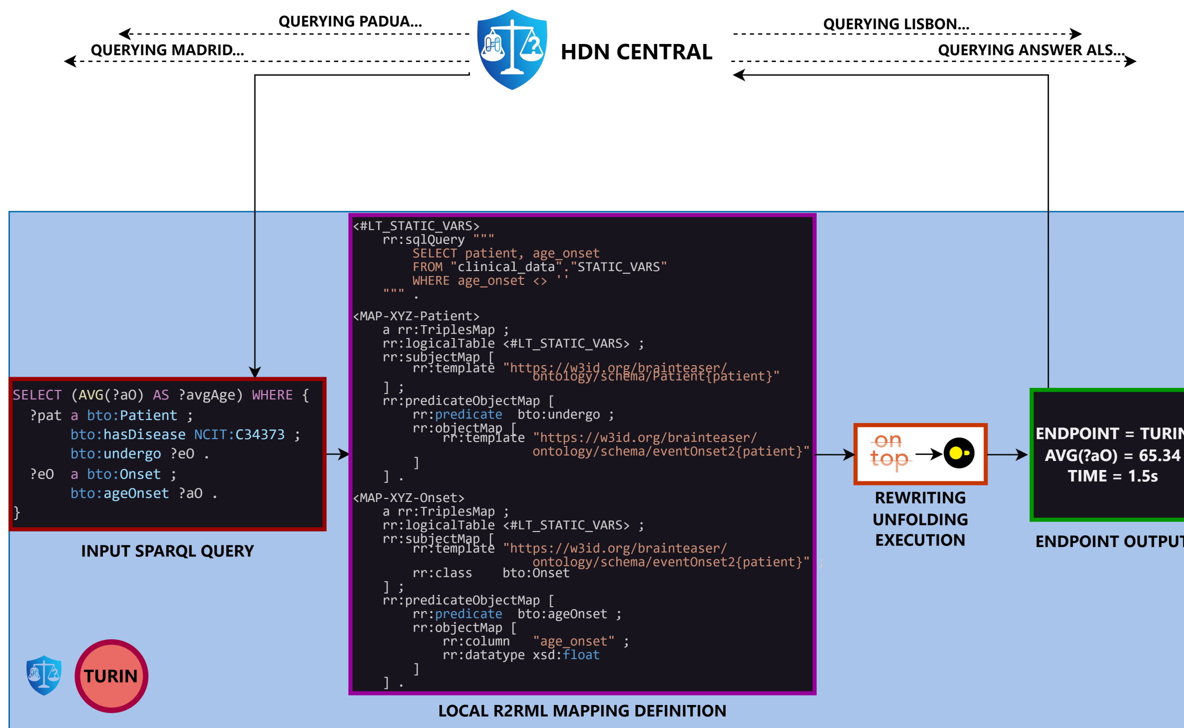


Figure 12: End-to-end HDN cyclic workflow for the L2 ALS onset-age query at the endpoint level. Starting from a catalog template, HDN Central dispatches the SPARQL query to all eligible endpoints, each of which rewrites it to SQL through Ontop, executes it over local clinical tables, and returns a scalar aggregate. HDN Central merges the partial results into a federated answer, together with provenance and timing metadata.

Output and observed timings. Table 4 reports, for this specific template, the effective sample size and the local mean age at onset per clinical endpoint, together with the global federated average. Across Lisbon, Madrid and Turin we obtain a combined cohort of $3{,}329$ ALS patients with non-missing onset age, with a mean around 64 years; Padua contributes an additional 45 ALS patients with a slightly younger onset distribution, yielding a global federated estimate of about 63.9 years over $3{,}374$ patients.

Table 4: Use case L2 (average ALS age at onset) – local and federated results computed over the clinical datasets.

Endpoint	#Patients with onset age	Mean age at onset
Lisbon (BRAINTEASER)	1306	62.19
Madrid (BRAINTEASER)	170	63.39
Turin (HEREDITARY/BRAINTEASER)	1853	65.34
Padua University Hospital	45	58.89

From the user perspective, the entire process appears as a single catalog query returning one scalar and a compact provenance summary (endpoints contributing, patient counts, disclosure level).

Using median per endpoint latencies, the slowdown versus a single endpoint is modest to moderate: $\times 1.32$ ($e{=}2$), $\times 1.63$ ($e{=}3$), $\times 1.39$ ($e{=}4$), $\times 1.75$ ($e{=}5$). The median coefficient of variation increases with the number of endpoints ($0.14 \rightarrow 0.22$), indicating slightly noisier latencies under larger federations, in line with endpoint heterogeneity (network conditions, hardware, coverage of individual templates). Overall, the ALS workload preserves the *compact core + short tail* profile: the majority of clinical/metadata templates (L0–L4 and the L2 onset-age use case) remain sub–half–second even at $e{=}5$, while a small set of slower L5-L6 queries raises the timings as federation grows.

Taken together, these results show that HDN keeps the ALS workload responsive under federation, while exposing a heterogeneous, multi-centre clinical landscape through a homogeneous semantic interface. The worked example of Listing 2 illustrates how a clinically meaningful statistic (average ALS onset age) can be obtained as a single federated query, without centralizing raw data and with clear privacy guarantees enforced by the disclosure-level framework.

Figure 13 shows a single–row grid (scale fixed) with one column per endpoint. Two patterns emerge: First, as endpoints grows, bars lift mildly for the majority of templates (coordination/union overhead), while a handful of L5-L6 queries elongate into a visible tail. Second, variability (error bars) stays compact for the fast core and widens in the tail, where endpoint heterogeneity contributes non–uniform latencies. The heaviest template is q13_L6, already non–trivial with just 1 endpoint ($\approx 4.15\,$s) and dominating the tail also in the subsequent configurations (≈ 7–$7.70\,$s). q12_L5 contributes as consistent second largest latency, with q09_L4 occasionally spiking (higher average standard deviation).

The boxplot in Figure 14 summarizes the same picture distributionally. Medians progress from 0.199s with 1 endpoint to 0.348s with 5 endpoints, with a non–monotone trend at 4 endpoints 0.276s, consistent with endpoint heterogeneity (some sources are lighter and/or not hit by all query templates). The upper data point reflect the tail dominated by q13_L6.

Using median per–endpoint latencies, the slowdown versus a single endpoint is modest to moderate: $\times 1.32$ ($e{=}2$), $\times 1.63$ ($e{=}3$), $\times 1.39$ ($e{=}4$), $\times 1.75$ ($e{=}5$). The median coefficient of variation increases with increasing number of endpoints ($0.14 \rightarrow 0.22$), indicating slightly noisier latencies under larger federations.

For most of the queries considered, the best observed latency occurs at 1 endpoint configuration; q12_L5 is the only case favoring 3 endpoints configuration. Relative to the per–query best measure, medians at endpoints=2,3,4,5 are respectively ($\times 1.32$), ($\times 1.44$), ($\times 1.46$), and ($\times 1.80$) slower. Overall, the ALS workload preserves the *compact core + short tail* profile: the majority of clinical/metadata queries remain sub–half–second even at ($e{=}5$), while a small set of slower queries rise the timings as federation grows.

Taken together, these results show that HDN keeps the ALS workload responsive under federation. Most clinical/metadata templates complete well below a second even at 5 endpoints; federation overheads raise medians by ($\approx 1.3\times$)–($1.8\times$) versus single–endpoint execution, with the tail concentrated in query templates with substantially larger result sets (levels L5–L6). Endpoints heterogeneity and partial coverage explain non–monotone medians and widening tails, while the execution layer remains robust.

3.2 Towards Machine Learning in HDN: a Case Study on the Cox model

HDN is the solution we are building in HEREDITARY to perform federated analytics. Given the goal of the project, a natural question that emerges is: how can HDN accommodate federated learning tasks? In a joint effort between WP2 and WP3, we are investigating state-of-the-art frameworks for federated learning (e.g. Flower): therefore, one option would be to integrate such frameworks in HDN. A major risk we can anticipate with such an option is that we may end up with two subsystems, one for federated analytics and one for federated learning, that co-exist with minimal integration, leading to challenges w.r.t. management of data privacy constraints or limiting the opportunities to exploit synergies for optimising analytics and learning execution. As an alternative, we consider another approach, where the HDN federated query and analytics architecture sets the foundation to perform machine learning tasks. To validate this idea, in this section we

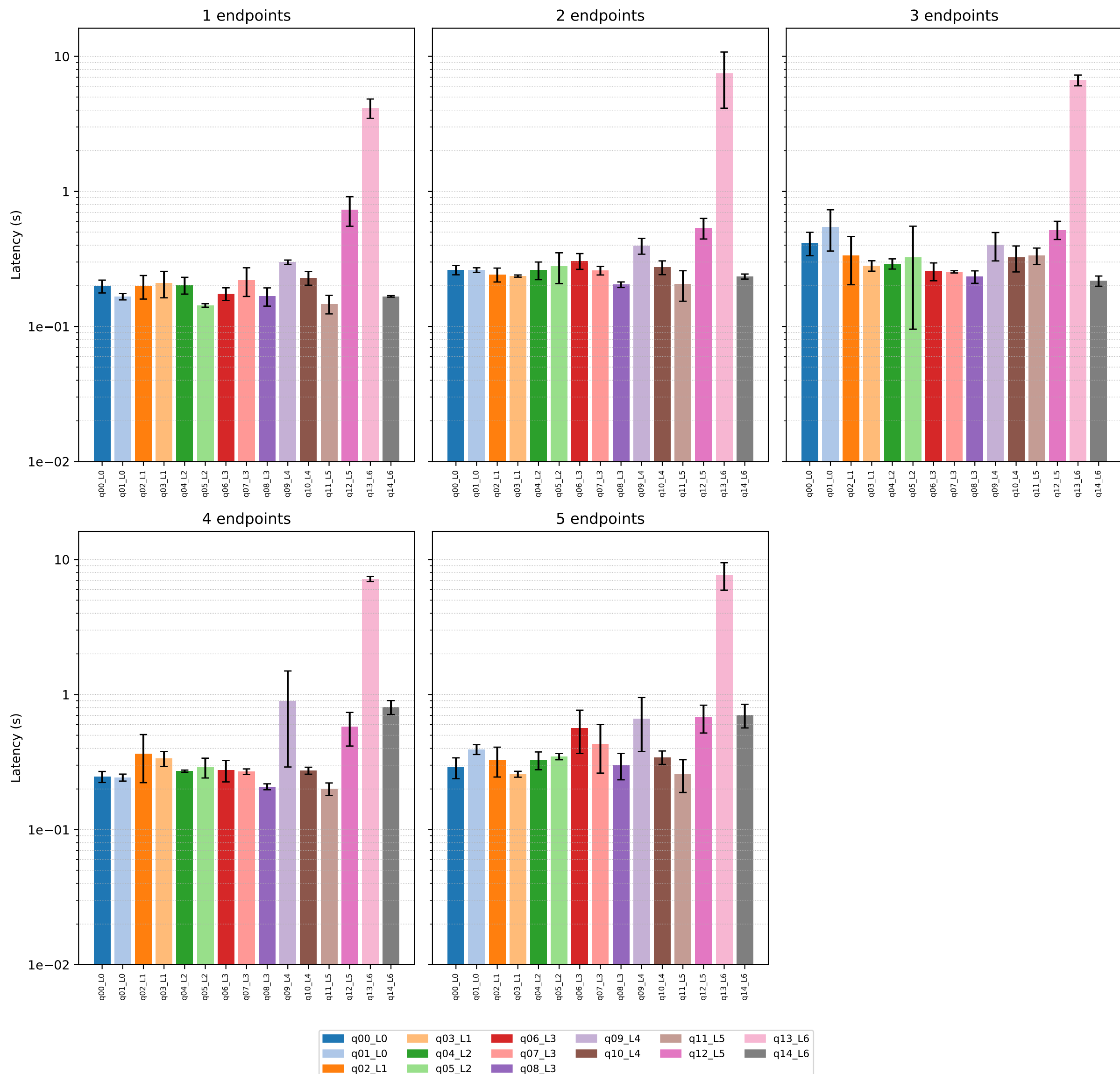


Figure 13: ALS use case — per–query mean latencies (logarithmic) at fixed scale and increasing federation size (1 to 5 endpoints). Error bars show standard deviation. Most templates remain sub–second across the grid; a small set forms the long tail as endpoints increase.

describe our initial efforts in writing machine learning algorithms through SQL statements, setting the basis for the design of machine-learning enabled HDN Central and HDN Endpoints to be developed in the next phase of the project.

We start with a brief analysis of the problem of creating local nodes (such as HDN endpoints) to perform machine learning tasks. A common approach to implement machine learning models is to rely on an *Extract-Transform-Load (ETL)* process. In this paradigm, data is first extracted from a source system (usually

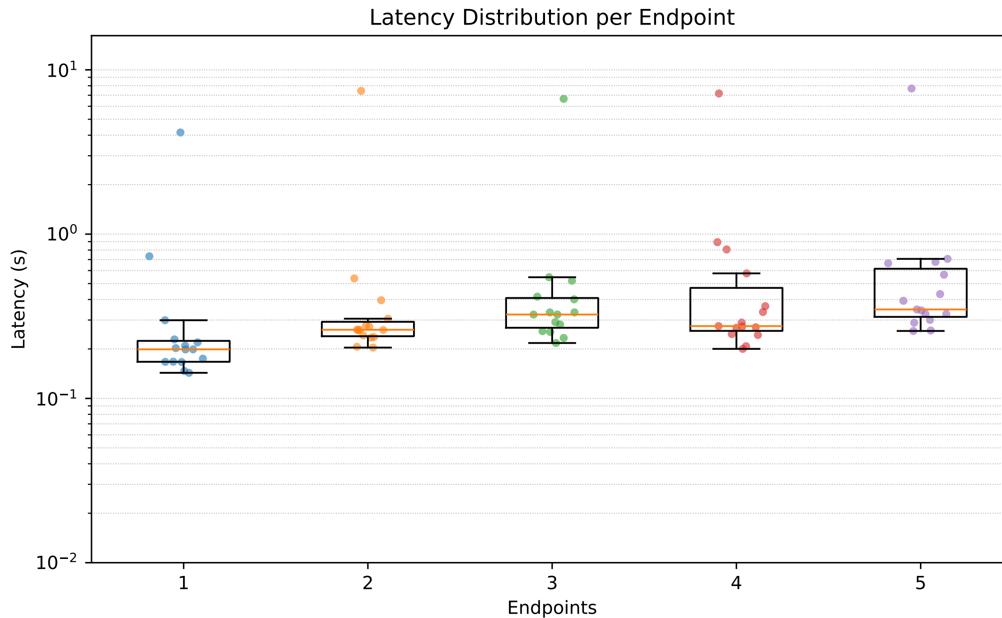


Figure 14: Latency distributions (log–y) across federation sizes. Boxes summarize per–query latencies; dots identifies individual queries latencies.

a DBMS), then transformed according to business logic, and finally loaded onto another storage system for reporting purposes. To implement ETL-based pipelines in HEREDITARY, owners would have to install multiple software frameworks, including those required machine learning, on the machines storing the patient data. This introduces two challenges. Firstly, HEREDITARY partners may need to undergo bureaucratic processes to install such software frameworks in machines that store sensitive patient data. Secondly, installing additional software frameworks in machines storing sensitive data increases the attack surface and the likelihood of exploitable bugs.

Moreover, when we consider the HDN architecture presented above, learning algorithms are generally more complex than analytics tasks, e.g. the average computation described in Section 3.1. This leads to two further challenges. Firstly, machine learning frameworks typically operate on data stored in main memory, which may necessitate additional hardware resources such as increased RAM or specialised processing units (e.g. GPUs). Secondly, the OBDA approach rewrites semantic queries into database queries. Existing OBDA solutions do not support the invocation of machine learning frameworks, creating a gap in current implementations.

To address these challenges, we want to implement machine learning algorithms directly within the DBMS, using SQL-based algorithms. SQL-based algorithms [7, 14, 18] use SQL queries to delegate most of the computation to the database engine; an external host program, written in a language of choice such as Python, acts as an orchestrator, issuing the queries, collecting intermediate results and combining them. Unlike ETL-like solutions, the extract step is eliminated, as the data remains in the DBMS, and the transform step is performed internally by the DBMS. An example is shown in Figure 15: the host program issues queries to the DBMS, processes resulting aggregates and issues new queries, until convergence.

SQL-based algorithms represent a particularly effective solution for a wide range of learning tasks within the HEREDITARY framework. By operating directly inside database engines, they naturally accommodate

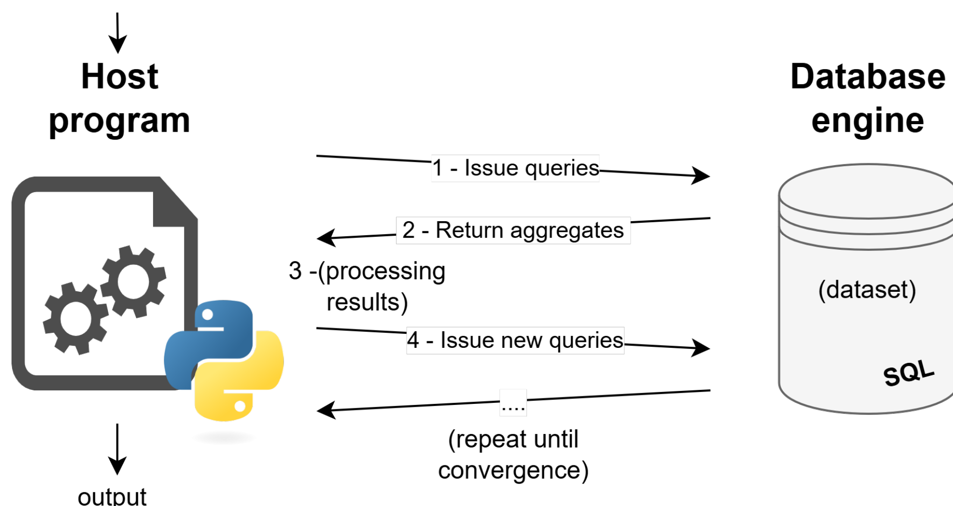


Figure 15: Example of a SQL-based algorithm.

heterogeneous data sources and schemas, allowing analytical workflows to be deployed across institutions without requiring data homogenization or migration. Their ability to scale beyond main memory and to exploit intra- and inter-query parallelism makes them well suited for large clinical and genomic datasets, while their execution benefits from decades of optimization in modern database management systems, resulting in robust and predictable performance. Beyond efficiency, SQL-based approaches offer important governance and security advantages. Since data remains within institutional databases and computations are pushed to where the data reside, the need to export sensitive information is minimized, significantly reducing privacy risks. At the same time, these algorithms can seamlessly leverage existing access control, auditing, and authentication mechanisms already enforced by the DBMS, aligning with privacy-by-design principles. Together, these characteristics make SQL-based learning a natural fit for federated analytics in HEREDITARY, as summarized in Table 5.

Table 5: Why SQL-based algorithms for HDN? A summary of benefits.

SQL-based benefit	Reason
Privacy/Security	data is processed inside the DBMS, exploiting its security layer (e.g. access control, encryption)
Scalability	leverages OLAP engines like DuckDB/Trino that work on bigger-than-memory datasets
Flexibility	adapts to heterogeneous sources, as long as they provide a SQL interface
Transparency	performs result aggregation and keeps track of provenance
Performance	relies on query engines which supports e.g. query optimization and parallel execution provided by the DBMSs

In the remainder of this section, we discuss the implementation of a machine learning model as a SQL-based algorithm. We chose to focus on the Cox model [11], a semiparametric model commonly used in the

medical domain. It is a type of regression used in survival analysis to examine how various factors influence the rate (or hazard) of an event occurring over time. When applied to survival datasets, the Cox model shows which elements are more influential to the survivability of ill patients, and thus it is useful for studying the impact of features.

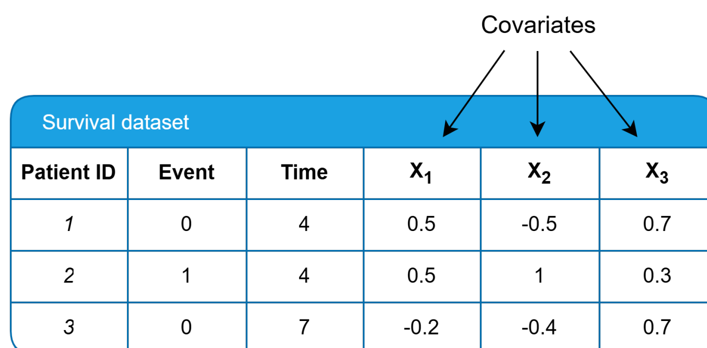
We start by providing the background on the Cox model in Section 3.2.1: we introduce the format of the input data—usually referred to as the survival dataset, and the model, including the learning procedure. We proceed by showing how to implement the model as a SQL-based algorithm. This SQL-based implementation serves as our foundation for running the Cox model in a distributed setting using existing distributed SQL engines. Finally, we discuss how we may adapt the distributed setting to the HDN network, showcasing open challenges and future directions.

We also provide a demo, available at `https://github.com/hereditary-eu/demo-federated-cox-trino`.

3.2.1 Background on the Cox Model

This section presents a formal definition of survival datasets, and then how they are used to build the Cox model.

Survival datasets. Survival datasets, like the one in Figure 16, model scenarios like medical trials, where patients either die or are censored (they interrupt their treatment or they end the trial). In Figure 16, we can observe different records, one for each patient. Each record is associated with a "Patient ID", an event (either 0—censored or 1—death), the time of that event. Finally, each record is associated with a set of covariates, for which the Cox model learns their effect on the survival of the patient.



Patient ID	Event	Time	X1	X2	X3
1	0	4	0.5	-0.5	0.7
2	1	4	0.5	1	0.3
3	0	7	-0.2	-0.4	0.7

Figure 16: Schema of a survival dataset. In the example, each record has a *Patient ID* and five other attributes: the final recorded *Event*, the *Time* at which the *Event* occurred, and three covariates X_1, X_2 and X_3

Let $\mathcal{S} = \{1, 2, \ldots, n\}$ be the set of patients (identified by an index value). A survival dataset is formally a list of tuples: $(\mathcal{S}, e, t, \boldsymbol{X})$, where:

- $e : \mathcal{S} \to \{0, 1\}$, where the value is 1 if the patient i died or 0 if the patient was censored (alive at the last check-up);
- $t : \mathcal{S} \to \mathbb{N}$ is the time at which the final registered event $e(i)$ occurred;
- $\boldsymbol{X} : \mathcal{S} \to \mathbb{R}^m$ is a vector with m features associated with patient i.

From the patient set $\mathcal{S}$, we define the failure patient set $\mathcal{D}_\mathcal{S} \subseteq \mathcal{S}$ as $\mathcal{D}_\mathcal{S} = \{\, i \in \mathcal{S} : e(i) = 1\}$, and the risk set $R_\mathcal{S} : \mathbb{N} \to \mathbb{P}(\mathcal{S})$ that associates a time instant with the set of indices having associated time equal to or greater than that time instant, i.e. $R_\mathcal{S}(\hat{t}) = \{\, i \in \mathcal{S} : t(i) \geq \hat{t}\,\}$.

The Cox model. Starting from a survival dataset, the Cox model assumes that the risk of a patient dying at a time $\hat{t}$ (hazard) follows the equation:

$$H(\hat{t} \mid \hat{\boldsymbol{X}}) = h(\hat{t}) e^{\boldsymbol{w}^\top \hat{\boldsymbol{X}}}, \tag{5}$$

where:

- $H(\hat{t} \mid \hat{\boldsymbol{X}})$ is the hazard, i.e., the probability of dying at time $\hat{t}$ for a patient with features $\hat{\boldsymbol{X}}$
- $h(\hat{t})$ is an unknown hazard baseline function; it represents the risk of dying at time $\hat{t}$ for a patient with all $\hat{\boldsymbol{X}} = \boldsymbol{0}$
- $\hat{\boldsymbol{X}}$ are the patient's features,
- $\boldsymbol{w}$ are unknown weights.

We are interested in learning which $\boldsymbol{w}$ better aligns with a given survival dataset; then, we can inspect the $\boldsymbol{w}$ to understand which features have the largest impact on patient survivability according to that dataset. No assumption is made about the hazard baseline $h(\hat{t})$: we prove that we can learn $\boldsymbol{w}$ independently.

In order to learn $\boldsymbol{w}$, we have to consider the likelihood function of $\boldsymbol{w}$. The likelihood function is the inverted version of a model, where weights become variables and vice versa. In our case, we invert Equation 5: the equation shows the probability of dying at a certain time given certain $\boldsymbol{w}$, by inverting that equation we can get the probability of $\boldsymbol{w}$ being a certain value, given that patients died at a certain time. Assuming probability independence between patients, we can express the likelihood as:

$$L(\boldsymbol{w}) = \prod_{i \in \mathcal{D}_\mathcal{S}} P_{\boldsymbol{w}}(e(i) = 1 \wedge t(i) = \hat{t}) = \prod_{i \in \mathcal{D}_\mathcal{S}} \frac{e^{\boldsymbol{w}^\top \boldsymbol{X}(i)}}{\sum_{j \in R_\mathcal{S}(t(i))} e^{\boldsymbol{w}^\top \boldsymbol{X}(j)}} \tag{6}$$

We now employ the Newton-Raphson method [1] to find the $\boldsymbol{w}$ that maximizes the likelihood. Compared to gradient descent, the Newton-Raphson method requires both the first and second order derivatives of the function, but it provides faster convergence and overall better asymptotic complexity [3]. To simplify computations, instead of maximizing the likelihood, we consider the dual problem of minimizing the negative logarithm of the likelihood. Starting from an initial guess $\boldsymbol{w}_0 = \boldsymbol{0}$, we update the coefficient vector at each iteration according to:

$$\boldsymbol{w}_{n+1} \leftarrow \boldsymbol{w}_n - \boldsymbol{H}_\ell(\boldsymbol{w}_n)^{-1} \nabla \ell(\boldsymbol{w}_n),$$

where:

- $\boldsymbol{w}_{n+1}$ are the new weights
- $\boldsymbol{w}_n$ are the previous weights
- $\boldsymbol{H}_\ell$ is the second order derivative of the negative log likelihood ℓ
- $\nabla \ell$ is the first order derivative of the negative log likelihood ℓ

The negative log likelihood and its derivatives can be written as:

$$\ell(\boldsymbol{w}) = -\log\left(L(\boldsymbol{w})\right) = -\sum_{i \in \mathcal{D}_\mathcal{S}} \left(\boldsymbol{w}^\top \boldsymbol{X}(i) - \log \sum_{j \in R_\mathcal{S}(t(i))} e^{\boldsymbol{w}^\top \boldsymbol{X}(j)} \right) \tag{7}$$

$$\nabla \ell(\boldsymbol{w}) = -\sum_{i \in \mathcal{D}_\mathcal{S}} \left(\boldsymbol{X}(i) - \frac{\sum_{j \in R_\mathcal{S}(t(i))} e^{\boldsymbol{w}^\top \boldsymbol{X}(j)} \boldsymbol{X}(j)}{s^{(0)}(t(i))} \right) \tag{8}$$

$$\boldsymbol{H}_\ell(\boldsymbol{w}) = \sum_{i \in \mathcal{D}_\mathcal{S}} \left(\frac{\sum_{j \in R_\mathcal{S}(t(i))} e^{\boldsymbol{w}^\top \boldsymbol{X}(j)} \boldsymbol{X}(j) \boldsymbol{X}(j)^T}{s^{(0)}(t(i))} - \frac{\sum_{j \in R_\mathcal{S}(t(i))} e^{\boldsymbol{w}^\top \boldsymbol{X}(j)} \boldsymbol{X}(j) \cdot \sum_{j \in R_\mathcal{S}(t(i))} e^{\boldsymbol{w}^\top \boldsymbol{X}(j)} \boldsymbol{X}(j)^T}{\left(s^{(0)}(t(i))\right)^2} \right) \tag{9}$$

Finally, we improve convergence by applying *step-halving*: when the likelihood does not improve after an iteration (i.e. $\ell\big(\boldsymbol{w}_{n+1}\big) \leq \ell\big(\boldsymbol{w}_n\big)$), we assign:

$$\boldsymbol{w}_{n+1}^{'} \leftarrow \frac{\boldsymbol{w}_{n+1} + \boldsymbol{w}_n}{2}$$

The *step-halving* procedure promotes a decrease in the objective at each iteration and thus improves convergence, especially in the early stages.

To sum up, Algorithm 1 depicts the learning algorithm for the $\boldsymbol{w}$ parameters of the Cox model. Here we can see the algorithm workflow explained above. In the first block (line 1-3) we initialize to zero the weights $\boldsymbol{w}$ and loss ℓ. We then proceed with the main loop (remaining lines, 4-20), where we impose a maximum number of iterations K. Inside the loop, we get the first-order (g) and second-order (H) derivatives (line 5) and apply the Newton-Raphson formula (line 6-7), obtaining w_{new}; we then check the loss ℓ_{new} on w_{new} (line 8):

- if the loss decreased since the previous iteration, it means we moved too far and missed the minimum, we need to perform step-halving and repeat the loop (lines 9-13);
- if the loss is stable, we deduce we have reached the minimum before iteration K, and thus we break the loop (lines 16-18);
- in other cases, we just update w and ℓ (lines 14-15, 19) and go to the next loop iteration.

After exiting the loop, we simply return w.

3.2.2 A SQL-based Implementation of the Cox Model

To provide a SQL-based implementation of the Cox model algorithm, we rewrite the two key functions $\texttt{compute_likelihood}(w)$ and $\texttt{compute_derivatives}(w)$ as SQL statements. The result is an algorithm that extracts only aggregates from the database, and does not expose privacy-protected data to the host.

Rewriting faithfully the definitions provided in Equations 7, 8 and 9, however, results in quadratic/exponential asymptotic complexity, leading to unfeasible computations even for small datasets. We identify two main issues: (1) the equation has a mix of iterations over the patients and iterations over time, and (2) at each

Algorithm 1: Fitting the Cox model through Newton-Raphson method with step-halving

```
    Input: maximum iterations K, tolerance ε, dimension p
    Output: estimated weights w

    // initial setup, set weights and loss to 0
 1  w ← 0 ∈ ℝ^p
 2  w_prev ← 0 ∈ ℝ^p
 3  ℓ ← 0

    // main iteration loop; at every step we apply Newton-Raphson formula
    // we try to improve w until convergence or maximum iteration reached
 4  for i ← 1 to K do
        // compute first and second order derivatives
 5      (g, H) ← compute_derivatives(w)
        // Newton-Raphson update formula
 6      δ ← H^{-1} g
 7      w_new ← w − δ

        // if likelihood is not improving, perform step halving
        // this happens when we overshoot and we go over the minimum
 8      ℓ_new ← compute_likelihood(w_new)
 9      if ℓ_new < ℓ then
10          w ← (w + w_prev) / 2
11          ℓ ← compute_likelihood(w)
12          continue
13      end

        // if likelihood improved, update w
14      w_prev ← w
15      w ← w_new
        // if ℓ is very close to ℓ_new, we are not improving:
        // we have reached convergence, exit
16      if |1 − ℓ_new/ℓ| < ε then
17          break
18      end
19      ℓ ← ℓ_new
20  end
21  return w
```

index i of the summation, we need to recompute as many of the $i-1$ previous elements as necessary, since each value may depend on each of the previous ones.

Existing implementations of the Cox model cope with these issues by exploiting dynamic programming [17], i.e., by choosing the right loop structure and caching intermediate results. This is possible thanks to the stateful nature of imperative programming, which makes it easy to mix loops and declare caching variables; the same approach is not directly reproducible in a declarative language such as SQL. In particular, any naive implementation that iterates through both patients and times needs a join later on, and this dramatically slows down the execution.

In order to optimize the implementation, we provide the mathematical rewriting of Equations 7, 8, and 9. We first define three auxiliary functions: $s^{(0)}(t)$, $s^{(1)}(t_q, a)$ and $s^{(2)}(t, a, b)$; we then use these three auxiliary

functions to redefine $\ell(\boldsymbol{w})$, $\nabla\ell(\boldsymbol{w})$ and $\boldsymbol{H}_\ell(\boldsymbol{w})$.

$$s^{(0)}(t) = \sum_{\hat{t}\in t(\mathcal{R}_\mathcal{S}(t_q))} \sum_{j\in T_{\mathcal{D}_\mathcal{S}}(\hat{t})} e^{\boldsymbol{w}^\top \boldsymbol{X}(j)} \tag{10}$$

$$s^{(1)}(t_q, a) = \sum_{j\in R_\mathcal{S}(t_q)} e^{\boldsymbol{w}^\top \boldsymbol{X}(j)} \boldsymbol{X}(j)[a], \tag{11}$$

$$s^{(2)}(t, a, b) = \sum_{j\in R_\mathcal{S}(t)} e^{\boldsymbol{w}^\top \boldsymbol{X}(j)} \boldsymbol{X}(j)[a]\boldsymbol{X}(j)[b], \tag{12}$$

$$\ell(\boldsymbol{w}) = -\sum_{i\in\mathcal{D}_\mathcal{S}} \left(\boldsymbol{w}^\top \boldsymbol{X}(i)\right) + \sum_{t_q\in t(\mathcal{D}_\mathcal{S})} \left(|T_{\mathcal{D}_\mathcal{S}}(t_q)| \cdot \log s^{(0)}(t_q)\right) \tag{13}$$

$$\nabla\ell(\mathbf{w})[a] = -\sum_{t_q\in t(\mathcal{D}_\mathcal{S})} \left(\sum_{i\in T_{\mathcal{D}_\mathcal{S}}(t_q)} X(i)[a] \;-\; \left|T_{\mathcal{D}_\mathcal{S}}(t_q)\right| \frac{\mathrm{s}^{(1)}(t_q,\, a)}{\mathrm{s}^{(0)}(t_q)} \right). \tag{14}$$

$$\mathbf{H}_\ell(\mathbf{w})[a, b] = \sum_{t_q\in t(\mathcal{D}_\mathcal{S})} \left|T_{\mathcal{D}_\mathcal{S}}(t_q)\right| \left(\frac{s^{(2)}\left(t_q, a, b\right)}{s^{(0)}\left(t_q\right)} - \frac{s^{(1)}\left(t_q, a\right) s^{(1)}\left(t_q, b\right)}{\left(s^{(0)}\left(t_q\right)\right)^2} \right). \tag{15}$$

Our optimized rewriting solves both issues mentioned above. Firstly, by isolating time-based iterations and patient-based iterations, we can perform two nested aggregations using *GROUP BY*, thus avoiding any join. Secondly, our rewriting is friendly to the SQL operator for cumulative sums, *SUM OVER()*. The *SUM OVER()* provides, for each row n, the summation of every row up to n. This computation is done in one pass, thus keeping the algorithm quasi-linear.

As an example, consider the second part of the likelihood in Equation 13,

$$\ell_2(\boldsymbol{w}) = \sum_{t_q\in t(\mathcal{D}_\mathcal{S})} \left(|T_{\mathcal{D}_\mathcal{S}}(t_q)| \cdot \log s^{(0)}(t_q)\right)$$

As stated above, we can rewrite it in SQL using two nested aggregations and the *SUM OVER()* operator.

```
WITH exp_wx_by_time AS (
  SELECT t, SUM(event) AS count_deaths, SUM(wx) AS sum_ewx
  FROM patient_with_wx
  GROUP BY t
),
cumulative_exp_wx_over_time AS (
  SELECT t,
         count_deaths,
         SUM(sum_ewx) OVER (ORDER BY t DESC) AS cumulative_sum_ewx
  FROM sums_by_time
)
SELECT SUM(count_deaths * LN(cumulative_sum_ewx)) AS l2
FROM cumulative_sums_over_time;
```

By applying our optimizations, we are able to achieve a quasi-linear asymptotic complexity for the likelihood for each of the n elements of $\nabla\ell(\boldsymbol{w})$ and for each of the n^2 elements of $\boldsymbol{H}_\ell(\boldsymbol{w})$.

3.2.3 Initial Evaluation of the SQL-based Implementation of the Cox Model

We evaluate the performance of our optimized implementation of the Cox model on part of the ALS dataset described in Section 3.1.1, specifically the BRAINTEASER part of the dataset [13]. After removing incomplete rows, we obtain a dataset with 1679 patients and 34 features. Selected features include both discrete features, such as sex, smoking, occupation and gene mutations, and continuous ones, such as weight and height. To study the scalability of our solution, we replicate the rows in the dataset, to preserve a realistic feature distribution and scale the data up to 10^6 data points. This lets us study the algorithm on a dataset that preserves the original distribution and therefore exhibits a similar convergence rate, while being arbitrarily large.

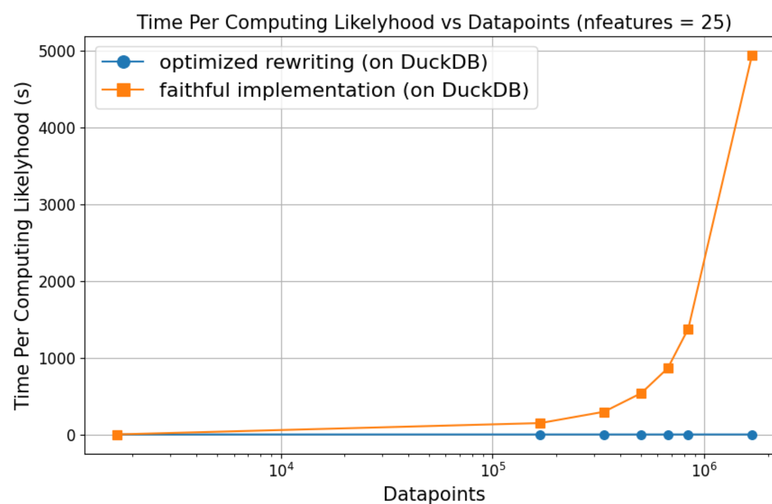


Figure 17: Benchmark: comparison between our implementation and the faithful one. Our implementation asymptotically outperforms the naive one.

We perform two experiments.

In the first experiment, whose results are shown in Figure 17, we compare two SQL-based implementations of the Cox model: one implements faithfully Equations 7, 8 and 9 introduced in Section 3.2.1, the other implements our rewriting, which is discussed in Section 3.2.2. Both queries are executed using DuckDB [22], a popular RDBMS engine tailored for numerical computations. DuckDB can run in-process, which means that it can be loaded as a library rather than as a system service; this makes it easy to avoid caching and keep the dataset in memory. We observe that our implementation outperforms the faithful one asymptotically: the curve of our implementation of the SQL-based algorithm maintains a running time below 10 seconds, while the curve of the faithful implementation grows faster than quadratically, resulting in a difference of up to three orders of magnitude. This experiment validates the optimisation we described in Section 3.2.2.

The second experiment compares our SQL-based algorithm, using DuckDB as our DBMS, with the implementation of the Cox algorithm in scikit-learn, a state-of-the-art implementation that relies on an ETL-like workflow, where data is first stored in RAM and then a Cox model is fitted on it. We measure only the time required by the computation itself, and not the time needed to load the dataset. In our SQL-based algorithm the dataset is stored in an in-memory instance of DuckDB, while data for the scikit-learn implementation is loaded from a CSV file using Pandas, the most popular library for in-memory dataframes in Python. Internally, scikit-learn stores each column of the dataframe as a NumPy array, a data structure which ensures efficient numerical computation while dealing with array operations [25]; NumPy provides arrays for fixed-size types,

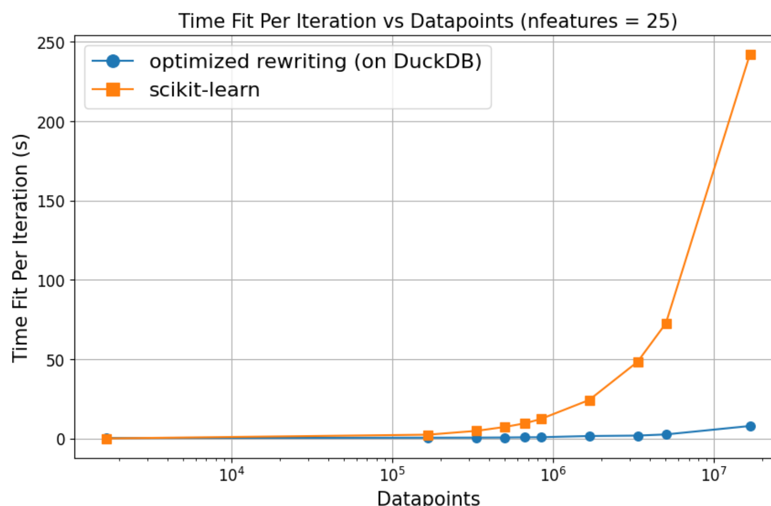


Figure 18: Comparison between our implementation and the state-of-the-art (scikit-learn). Our implementation is up to two order of magnitude faster than the state-of-the-art.

and is able to compile multiple algebraic operations together using a just-in-time compiler: this makes NumPy computations almost as fast as carefully hand-crafted low-level ones. As detailed in Figure 18, this second experiment reveals that our SQL-based algorithm is faster than scikit-learn up to two orders of magnitude. A measurement on our SQL-based implementation reveals that less than 0.1% of the total time is employed to combine the query results, and thus almost the totality of the computation happens inside the DBMS. Considering that our optimized algorithm and the scikit-learn one have the same theoretical asymptotic complexity, the comparison suggests that a DBMS engine such as DuckDB, as expected, is able to perform better internal optimizations on a SQL query than the imperative counterpart in Python/Numpy. While an imperative language offers little rooms for internal planning and optimisations, the declarative nature of SQL gives more freedom to the DBMS engine. Engines such as DuckDB are able to schedule and rearrange computations, perform computations in parallel, process chunks of data as one and deal with big data using cost-estimation strategies. These optimizations, especially when performed on big datasets, could be crucial to ensure that the computation is distributed efficiently across multiple *Central Processing Unit (CPU)* cores, and correctly employs memory caches. While libraries such as NumPy are able to speed up chains of algebraic operations on arrays, they lack the bigger picture, miss a query planner and miss years of optimizations on big data analytics, and this possibly explains why we measured such a speed difference in our benchmark.

We also want to stress that in both experiments, the dataset could fit in memory; if that was not the case, in the second experiment, scikit-learn would have just crashed, whereas DuckDB would have been successfully able to store the exceeding data on disk and complete the computation, albeit with a slowdown depending on the disk speed.

3.2.4 Distributed SQL-based Implementation of the Cox Model

Usually, SQL-based algorithms are executed on single database engines, but we can obtain a distributed version thanks to *Distributed SQL Engines (DSEs)* [21]. DSEs are engines that expose a single SQL-compliant endpoint and, under the hood, they transform the input query into local queries to be distributed to a network of database engines, and aggregate the answers. The redistribution happens automatically: the DSE acts as

a black box, employs a best-effort policy to minimize data movement, and uses optimized algorithms to perform the distribution of standard SQL operators. An example of DSEs is Dremio, which we used to implement the D3.1 architecture discussed in Section 2.1.

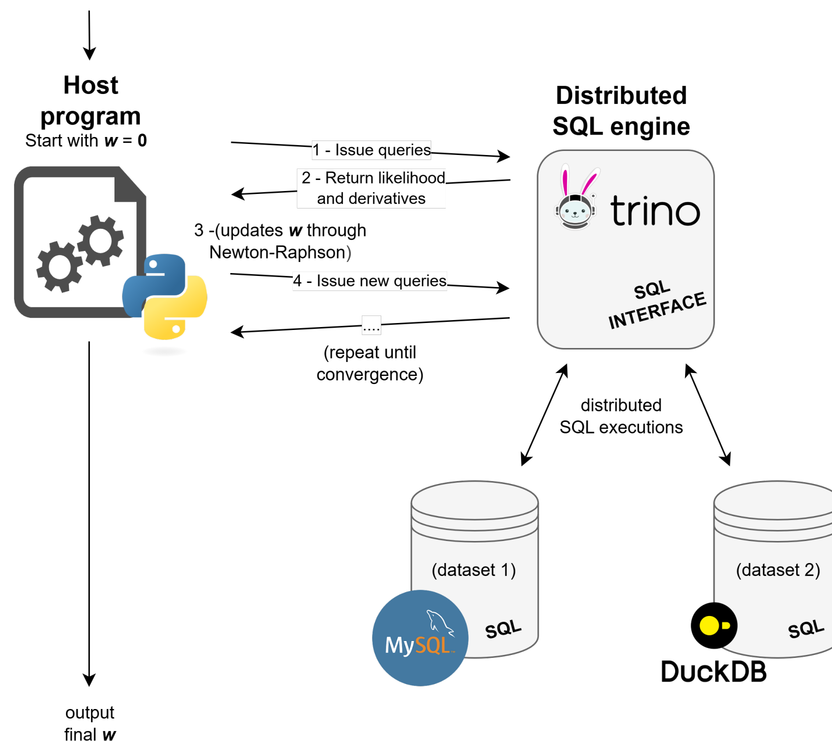


Figure 19: Architecture of a SQL-based implementation of the Cox model algorithm in a distributed environment. A host program issues queries to a SQL endpoint, receives the results, and postprocesses them. Compared to Figure 15, the single database engine has been replaced by a DSE. Under the hood, the DSE issues queries to two different database engines, and collects the results.

Currently, we are testing our distributed Cox model using Trino[10], an open source state-of-the-art DSE. The architecture comprises four key components: the host program (1.), which maintains its role as previously described in Section 3.2.2, but now communicates with a DSE; the DSE itself (2.), which replaces the standard database engine and offers a SQL-endpoint; and finally, two distinct data sources (3. and 4.) connected to the DSE: one implemented via a MySQL database engine and the other using DuckDB.

The algorithm flow proceeds as follows: the host program follows Algorithm 1; when the host program has to run compute_likelihood(w) or compute_derivatives(w), it issues a SQL query to the Trino DSE (the query is crafted following our rewriting, which is explained in Section 3.2.2). Trino takes the query and creates new sub-queries that are dispatched to the two underlying data sources; then, Trino collects the results, aggregates them, and sends the aggregated results back to the host program. The host program can now update the w and proceed with its computation. The process is repeated until convergence.

It is worth noting that, in this scenario, every complexity regarding data location and configuration has already been removed from the host program and delegated to Trino instead. This creates a clear role

[10]Trino: https://trino.io/

division between researchers, who should only care about crafting the host program, and maintainers of the data infrastructure.

3.2.5 Integration with HDN and Future Challenges

Our experiments on the SQL-based implementation of the Cox model (Section 3.2.3) and our considerations on how the model can work in a distributed setting (Section 3.2.4) indicate that the idea of building machine learning workflows on top of a federated query engine like HDN is feasible and worth further investigation. The SQL-based implementation presented in Section 3.2.2 separates the learning process into two components: a host program that orchestrates the submission of the SQL queries, and a DBMS that executes them.

The host program should be accommodated in HDN Central: on user request, this component should generate a set of semantic queries to be executed in the HDN Endpoints. Therefore, the first challenge we envision is whether we can represent machine learning algorithms in SPARQL-based implementations on top of HERO. Expressing machine learning algorithms through semantic queries may be hindered by lack of operators that are available in SQL but not in SPARQL (e.g. `SUM OVER`). A possible solution is to extend SPARQL with such operators, or to study alternative ways to express the algorithms in standard SPARQL, which may lead to a higher number of queries.

Another challenge concerning the orchestration of these algorithms by HDN Central relates to privacy. Currently HDN considers privacy for each query, but in the context of a machine learning algorithm, this may lead to undesired behaviours. A possible solution is to determine a common privacy level for all the semantic queries that compose the machine learning algorithm, and decide whether the algorithm can be executed successfully.

The execution of the semantic queries composing the SPARQL-based learning algorithm should be delegated to the HDN Endpoint. Considering HDN Endpoints based on the OBDA paradigm, as the ones discussed in Section 3.1, we observe two challenges. The first relates to the different availability of SPARQL and SQL operators described in the previous paragraph. We may need to extend existing OBDA engines to support the rewriting of new operators we add to SPARQL.

The second challenge relates to how OBDA works: in the general setting, a semantic query is rewritten in a set of SQL queries, i.e. there is a one-to-many mapping. However, similarly to a SQL-based algorithm, a SPARQL-based algorithm is a set of SPARQL queries. Therefore, an HDN endpoint will receive many SPARQL queries, which need to be transformed into SQL queries, i.e. there is a many-to-many mapping. While a naive solution is to consider each SPARQL query individually, the OBDA engine may consider the whole workload to generate a set of SQL queries that optimise the execution of the learning algorithm.

The challenges we presented in this section are examples of challenges we aim to investigate in the next period of the project, with the goal of enabling machine learning algorithms in HDN.

3.3 Public Datasets Supply for HEREDITARY project Needs

This section describes the integration of Ontotext's LLDI within the HEREDITARY ecosystem and its role in supporting the federated analytics objectives of the HDN. While the HDN focuses on executing privacy-preserving semantic queries across distributed institutional data, LLDI provides the complementary data management and publication infrastructure required to curate, standardize, and expose biomedical reference datasets in a FAIR-compliant manner. LLDI is a solution not only for easy deployment of the datasets to a specific GraphDB instance, but also for keeping them up-to-date. Each of the datasets is loaded in a different named graph which allows its smooth management in a transactional way, All the datasets from a specidic domain are interconnected and the references are aligned and this provides their interoperability out-of-the-box. Such approach is very beneficial for projects where data are heterogeneous and are coming

from different sources because it provides a level of harmonization and data links to external data sources - public or proprietary, given they are RDF-ized and supplied by the LLDI. Such way these datasets become a glue, a data connecting vessel. The product itself consists of two different parts, described further in the section:

- A centralized LLDI server running on Ontotext infrastructure. The FAIR-ification workflows are intregrated in this server. They transform datasets to RDF format and also generate datasets metadata and import files for datasets updates at customers. repositories
- Data Loader application which runs on client side. This application is connected to the client instance of GraphDB. It is fully configurable. LLDI also provides Data Catalogue which is an additional application for easy browsing available datasets and their metadata. It helps users to decide which datasets they need for their projects.

The section is structured as follows: first, an overview of the LLDI platform and its data processing workflows is presented; next, the Data Catalogue and Data Loader components are described; finally, the biomedical datasets made available through LLDI for HEREDITARY use cases are introduced.

3.3.1 LinkedLifeData Inventory

The LLDI is an Ontotext's automated platform for managing, transforming, and publishing biomedical data into standard *Resource Description Framework (RDF)* format, serving as a core component of *Knowledge Graph (KG)* development and significantly improving data FAIRness. LLDI workflows, orchestrated by Apache Airflow on Kubernetes, handle both public and proprietary datasets. Data not originally in RDF are transformed according to a target semantic model. Each pipeline includes an automated validation stage, followed by the generation of *Vocabulary of Interlinked Data (VoID)* and *Data Catalog Vocabulary (DCAT)* metadata vocabularies for semantic documentation and discoverability. This metadata is integrated into the LLDI Catalogue. Finally, mapping files are produced to generate links to external datasets, and the new artifact version is published to a central secure storage location, with its download endpoint referenced in the Data Catalogue. Importantly, the LLDI contributes to HDN by providing semantically harmonized biomedical resources that can be consumed by federated analytics workflows. Since LLDI publishes FAIR-compliant RDF data, the resulting datasets can be queried semantically across heterogeneous and independently evolving data sources while preserving the decentralized and privacy-preserving design principles of HDN. More details about FAIR-ification processes are provided in deliverable D3.4 FAIRification of participating data resources, section 6.5.

3.3.2 Data Catalogue

The semantic Data Catalogue is a FAIR-compliant application designed to ensure that datasets are Findable, Accessible, Interoperable, and Reusable. Using metadata standards such as DCAT and VoID, the catalogue precisely describes datasets and the relationships between them in a machine-readable format. The application enables dataset discovery through metadata-driven navigation, SPARQL endpoints, and RDF-based querying, allowing users to explore inter-dataset relationships and linkage. Linksets improve interoperability by explicitly defining semantic correspondences between resources originating from different datasets. Figure 20 shows the data catalogue search screen which allows comprehensive searches using various criteria and also faceted filtering on dataset types, frequency of updates, license types, and so on. The catalogue also provides configurable dashboards for metadata visualization, including schema overviews (classes and properties), triple counts, and diagrams illustrating cross-dataset connectivity.

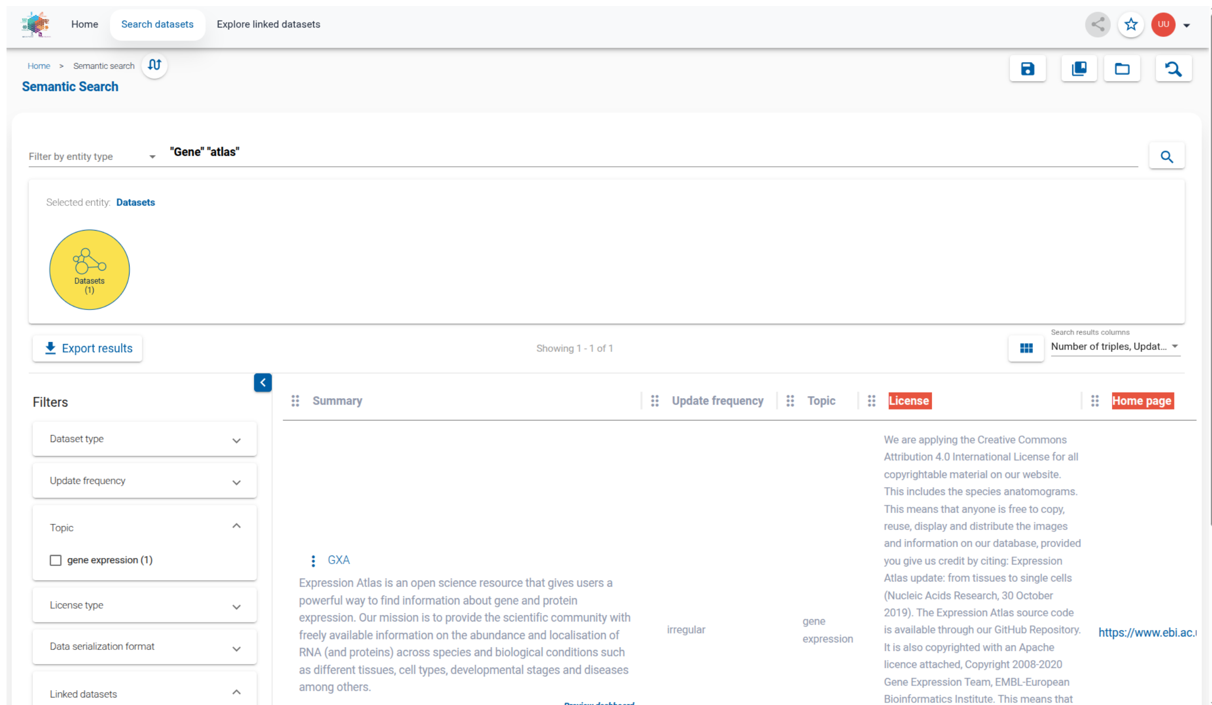


Figure 20: Searching for datasets in the data catalog

3.3.3 Data Loader

The Data Loader (also functioning as a data synchronisation and update tool) provides a client interface that allows users to inspect, manage, and update locally stored datasets to the latest available version published in the central LLDI metadata repository. Each user (client site) has its own user account in the LLDI and subscription for one or more datasets. Usong the shown interface users can view subscribed datasets, compare version states, and initiate or cancel updates. A screenshot of the dashboard is shown on Figure 21.

For each of the subscribed datasets the user have to provide a local configuration (see below) in order to use it. As is shown on the Figure 21, on this screenshot only the first two datasets are configured and the corresponding information about their current version, its publishing date, size, etc. and also the update statuses is displayed. Data Loader communicates with the central LLDI repository about subscribed datasets and receives data about their new versions and such way decides are updates necessary or not. This process is fully automated and users do not need to check for updates.

The update workflow includes:

1. Downloading the required dataset version .
2. Clearing the corresponding named graph in the local GraphDB repository.
3. Loading the new dataset into a named graph derived from the DCAT metadata.
4. Removing the locally downloaded files.

This process is performed in a transactional manner and ensures that local repositories remain aligned with the authoritative dataset version. While users must create and configure local repositories, the Data

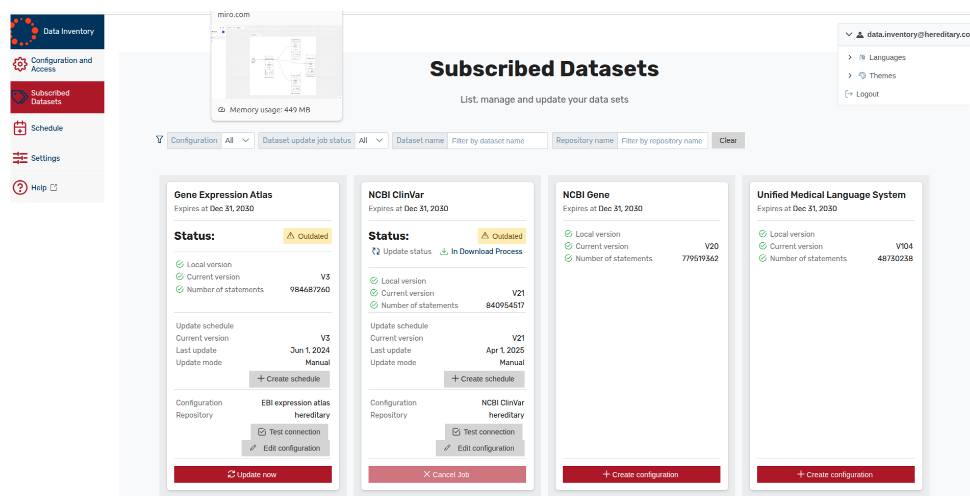


Figure 21: Data Loader Dashboard of the subscribed datasets from Linked Life Data Inventory

Loader automatically manages system-level repositories responsible for configuration storage, automatic update scheduling, and update status tracking.

Data Loader Features:

1. **Configurations.** Users can view, create, edit, and delete dataset and repository configurations. Credentials required for connecting to GraphDB and the Data Loader service are stored securely. A valid configuration is required before initiating updates.

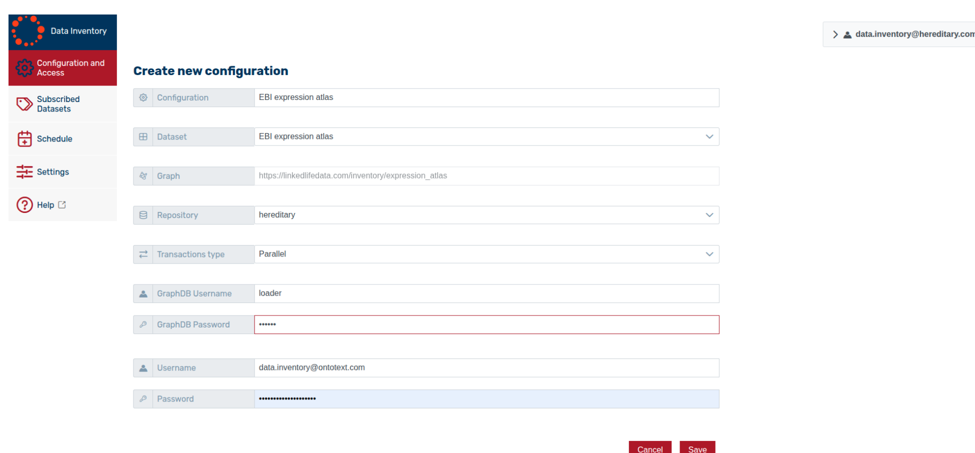


Figure 22: Creating and Configuration of Data Catalog from LinkedLife Data Inventory

Dataset Management: As shown on Figure 22 subscribed datasets are displayed as cards with configured datasets prioritized. Users may filter datasets, view update status (e.g., outdated), and com-

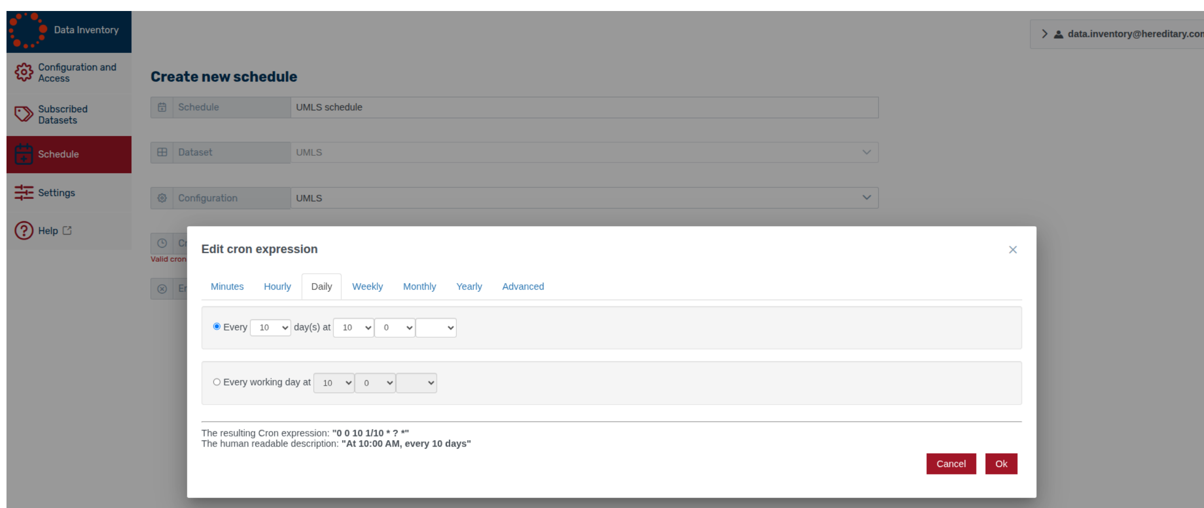


Figure 23: Configuring time schedule for a dataset updates

pare installed vs. available versions. Updates may be triggered manually or scheduled automatically. Failed updates are handled via configurable retry logic, and detailed status codes (e.g., IN_QUEUE, UP_TO_DATE, SUCCESSFULLY_CLEANED_UP) are recorded and displayed.

2. **Schedules.** Users may configure automated update schedules using Schedules functionality, shown in Figure 23. It can be defined either by providing cron expressions or using a convenient dialog. This functionality is especially useful when the data sources are published irregularly and when there are no custom changes to datasets after their updates. Schedules can be created in a disabled state for later activation. Once active, the Data Loader automatically initiates dataset updates according to the defined schedule. If there is a new version published, the update process is initiated.

3.3.4 Data Catalog Datasets

The datasets currently identified as relevant to the HEREDITARY project are listed below. Detailed descriptions of the corresponding FAIRification workflows are provided in **D3.6 – FAIRification Workflows.**

Expression Atlas Expression Atlas is an open-science resource supporting the exploration of gene and protein expression across tissues, cell types, and a wide range of biological conditions. It was included in LLDI especially for purposes of HEREDITARY project. Publisher: EMBL – European Bioinformatics Institute (EMBL-EBI). [11]

NCBI Gene NCBI Gene is a curated database that provides comprehensive information on known and predicted genes across species, including nomenclature, functional annotation, genomic context, and links to supporting resources. Publisher: National Center for Biotechnology Information (NCBI), U.S. National Library of Medicine. [12]

[11] https://www.ebi.ac.uk/gxa/home
[12] https://www.ncbi.nlm.nih.gov/gene

ClinVar ClinVar is a publicly accessible archive that aggregates information about human genetic variants and their clinical significance, including associated phenotypes and supporting evidence. Publisher: National Center for Biotechnology Information (NCBI), U.S. National Library of Medicine. [13]

UMLS The *Unified Medical Language System (UMLS)* integrates major biomedical terminologies, vocabularies, and coding standards, providing a unified semantic framework that supports interoperability across biomedical information systems. Publisher: National Library of Medicine. [14]

4 Milestone Verification

This deliverable verifies Milestone 8: "Federated workflow execution methods" by providing a detailed account of the design, implementation, and experimental validation of the HDN and its federated workflow execution engine. It describes the architecture of HDN Central and HDN Endpoints, clarifies privacy-preserving query orchestration using HERO, and outlines how queries are defined, dispatched, and results aggregated across a network of participating centers. Through extensive benchmarking—comparing HDN to the state-of-the-art OBDF federation stack (proposed and implemented in D3.1) on the HDN workload and ALS clinical data—the deliverable demonstrates HDN's robust performance, scalability, and privacy compliance. Median speedups ranging from 1.5 to 4.3, reduced federation latency penalties, and support for multimodal data integration substantiate successful milestone completion.

Additionally, the deliverable presents use-case studies that underline the federation engine's capacity: distributed querying on ALS datasets, the implementation of a distributed Cox model using SQL-based algorithms, and successful ingestion from external linked data portals. Figures, tables, and workflow diagrams document data preparation, architectural configuration, privacy management, and federated analysis outcomes. Together, these results give evidence that the federated workflow execution methods have been implemented and verified, meeting all criteria specified for milestone achievement within WP3.

5 Conclusions and Remarks

This document presents the first release of HDN, the HEREDITARY federated workflow execution engine. HDN allows for the definition and execution of queries using the HERO ontology. Building queries based on an ontology rather than on the data schema allows users to define their queries without needing to know the data schema of each data provider. From the perspective of the data providers, the HDN provides a privacy assurance mechanism that allows them to share the data.

The collective evidence across our benchmarks and use-case studies shows that HDN offers a distinctly superior foundation for federated analytics. HDN can sustain substantially larger data volumes and more endpoints without sacrificing responsiveness or predictability due to its consistently lower latencies, weaker sensitivity to federation size and more favorable scaling profile. In both synthetic stress tests and the ALS clinical workload, HDN delivers robust sub-second performance for the vast majority of queries while sharply reducing tail latencies compared to the legacy OBDF stack. Moreover, its ability to interoperate with heterogeneous data sources—ranging from HERO-based clinical metadata to large linked-data inventories—underscores its flexibility as an execution engine that can replace monolithic ETL pipelines with distributed, privacy-preserving

[13] https://www.ncbi.nlm.nih.gov/clinvar/
[14] https://www.nlm.nih.gov/research/umls/index.html

computation. Together, these results confirm that HDN is not only faster but also more scalable, reliable, and extensible, making it a strong architectural choice for modern federated workflows.

Moreover, the three use cases shows the applicability of the HDN to different federated data analytics scenarios. Firstly, the ALS workload, built on the HERO ontology and federated over five heterogeneous endpoints, demonstrated that even complex clinical metadata queries, while preserving privacy, remain largely sub-second. Secondly, the SQL-based Cox model shows that it is possible to write machine learning algorithms through queries, and this can be a viable solution to extend HDN to support both analytics and machine learning workloads. Finally, the successful ingestion of Ontotext's LLDI via the HDN catalog showcases how the network can seamlessly bridge structured, linked-data sources with its existing federated query infrastructure, enabling downstream analytics while preserving access control.

Future work will focus on hardening HDN's performance, scalability, and privacy guarantees while broadening its analytical capabilities. We plan to investigate adaptive query planning, dynamic caching, and elastic scaling to support larger federations and more heterogeneous endpoints. Moreover, we plan to deploy lightweight AI models locally at participating sites, enabling federated inference and classification under a unified governance model that leverages transfer learning and zero-shot knowledge graph embedding techniques, thanks to federated *Knowledge Graph Embeddings (KGE)* approaches such as FedE, FedR, and FedLU (to be integrated into the WP4 enrichment pipeline). In particular, we plan to implement the distributed Cox model over HDN, as well as other learning algorithms implemented in query languages. In addition, we aim to add bioimage feature extraction algorithms to enable cross-modal queries at the patient level. Privacy controls will be refined and verified against GDPR[15] and emerging AI governance requirements, with explicit attention to resilience against privacy attacks, so partners retain control over what is shared while the network scales its analytical capabilities. Finally, we will formalize the FAIRification of the catalog, integrate Beacon interoperability.

Finally, the current definition of levels L1–L6 provides a semantic contract between HDN Central and endpoints, but further work is needed to refine and operationalise this model from a privacy-engineering perspective. In particular, future work will have to verify how local endpoints implement and honour these levels in practice, and to specify the technical and organisational safeguards associated with each level. This includes:

- Privacy-preserving mechanisms such as differential privacy.
- Access control, authentication, and fine-grained authorisation policies aligned with institutional roles and purposes.
- Logging and audit trails for query execution, including data minimisation checks and enforcement of disclosure limits.
- Encryption and synthetisation of data in transit and at rest, especially for L4–L6.
- Systematic *Data Protection Impact Assessments (DPIA)* for higher-risk queries (in particular at L4–L6) and alignment with AI Act requirements for high-risk or automated analytical workflows.

From a compliance standpoint, a structured documentation and governance framework will also be required per privacy level, covering elements such as: the legal basis for processing; ethics committee approvals; data processing agreements (including joint controller arrangements where relevant); patient information and rights-management procedures (including opt-out); documented technical safeguards for aggregation and anonymisation; DPIAs for L4–L6; consent forms and re-identification risk analyses for L5–L6; audit logs and access records; international data transfer agreements where applicable; and AI system documentation for advanced analytics and automated decision-making. These aspects will be developed in collaboration

[15] https://eur-lex.europa.eu/eli/reg/2016/679/oj?locale=en

with clinical partners, legal teams, and data protection officers as part of future work on the HDN privacy framework.

References

[1] Akram, S. and Ann, Q. U. (2015). Newton raphson method. *International Journal of Scientific & Engineering Research*, 6(7):1748–1752.

[2] Arenas-Guerrero, J., Pérez, M. S., and Corcho, Ó. (2023). LUBM4OBDA: benchmarking OBDA systems with inference and meta knowledge. *J. Web Eng.*, 22(8):1163–1186.

[3] Battiti, R. (1992). First- and second-order methods for learning: Between steepest descent and newton's method. *Neural Comput.*, 4(2):141–166.

[4] Baxi, E. G., Thompson, T., Li, J., Kaye, J. A., Lim, R. G., Wu, J., Ramamoorthy, D., Lima, L., Vaibhav, V., Matlock, A., Frank, A., Coyne, A. N., Landin, B., Ornelas, L., Mosmiller, E., Thrower, S., Farr, S. M., Panther, L., Gomez, E., Galvez, E., Perez, D., Meepe, I., Lei, S., Mandefro, B., Trost, H., Pinedo, L., Banuelos, M. G., Liu, C., Moran, R., Garcia, V., Workman, M., Ho, R., Wyman, S., Roggenbuck, J., Harms, M. B., Stocksdale, J., Miramontes, R., Wang, K., Venkatraman, V., Holewenski, R., Sundararaman, N., Pandey, R., Manalo, D.-M., Donde, A., Huynh, N., Adam, M., Wassie, B. T., Vertudes, E., Amirani, N., Raja, K., Thomas, R., Hayes, L., Lenail, A., Cerezo, A., Luppino, S., Farrar, A., Pothier, L., Prina, C., Morgan, T., Jamil, A., Heintzman, S., Jockel-Balsarotti, J., Karanja, E., Markway, J., McCallum, M., Joslin, B., Alibazoglu, D., Kolb, S., Ajroud-Driss, S., Baloh, R., Heitzman, D., Miller, T., Glass, J. D., Patel-Murray, N. L., Yu, H., Sinani, E., Vigneswaran, P., Sherman, A. V., Ahmad, O., Roy, P., Beavers, J. C., Zeiler, S., Krakauer, J. W., Agurto, C., Cecchi, G., Bellard, M., Raghav, Y., Sachs, K., Ehrenberger, T., Bruce, E., Cudkowicz, M. E., Maragakis, N., Norel, R., Van Eyk, J. E., Finkbeiner, S., Berry, J., Sareen, D., Thompson, L. M., Fraenkel, E., Svendsen, C. N., and Rothstein, J. D. (2022). Answer ALS, a large-scale resource for sporadic and familial ALS combining clinical and multi-omics data from induced pluripotent cell lines. *Nature Neuroscience*, 25(2):226–237.

[5] Beutel, D. J., Topal, T., Mathur, A., Qiu, X., Parcollet, T., and Lane, N. D. (2020). Flower: A friendly federated learning research framework. *CoRR*, abs/2007.14390.

[6] Bizer, C. and Schultz, A. (2009). The berlin SPARQL benchmark. *Int. J. Semantic Web Inf. Syst.*, 5(2):1–24.

[7] Blacher, M., Giesen, J., Laue, S., Klaus, J., and Leis, V. (2022). Machine learning, linear algebra, and more: Is SQL all you need? In *CIDR*. www.cidrdb.org.

[8] Calvanese, D., Cogrel, B., Komla-Ebri, S., Kontchakov, R., Lanti, D., Rezk, M., Rodriguez-Muro, M., and Xiao, G. (2017). Ontop: Answering SPARQL queries over relational databases. *Semantic Web*, 8(3):471–487.

[9] Cazzaro, M., Gut, I. G., Menotti, L., Rueda, M., and Silvello, G. (2025). HERO-Genomics: An ontology for integration and access of multicenter genomic data. In *SWAT4HCLS*.

[10] Cazzaro, M., Menotti, L., Primov, T., Rueda, M., and Silvello, G. (2024). Deliverable 3.1: Ontology and federated execution methods. EU Project Deliverable D3.1, Hereditary Project.

[11] Cox, D. R. (1972). Regression models and life-tables. *Journal of the Royal Statistical Society: Series B (Methodological)*, 34(2):187–202.

[12] Faggioli, G., Menotti, L., Marchesin, S., Chiò, A., Dagliati, A., de Carvalho, M., Gromicho, M., Manera, U., Tavazzi, E., Nunzio, G. M. D., Silvello, G., and Ferro, N. (2024). An extensible and unifying approach to retrospective clinical data modeling: the brainteaser ontology. *J. Biomed. Semant.*, 15(1):16.

[13] Faggioli, G., Menotti, L., Marchesin, S., and et al. (2025). The brainteaser datasets: Clinical, wearable and environmental data for als & ms progression modeling. *Scientific Data*, 12:Article 1854.

[14] Giesser, P., Stechschulte, G., da Costa Vaz, A., and Kaufmann, M. (2021). Implementing efficient and scalable in-database linear regression in SQL. In *IEEE BigData*, pages 5125–5132. IEEE.

[15] Gu, Z., Calvanese, D., Panfilo, M. D., Lanti, D., Mosca, A., and Xiao, G. (2024). OBDF: OBDA + data federation - extended abstract. In *ICDEW*, pages 381–383. IEEE.

[16] Guo, Y., Pan, Z., and Heflin, J. (2005). LUBM: A benchmark for OWL knowledge base systems. *J. Web Semant.*, 3(2-3):158–182.

[17] Kalbfleisch, J. D. and Schaubel, D. E. (2023). Fifty years of the cox model. *Annual Review of Statistics and Its Application*, 10(1):1–23.

[18] Kaufmann, M., Stechschulte, G., and Huber, A. (2021). Efficient and accurate in-database machine learning with SQL code generation in python. *CoRR*, abs/2104.03224.

[19] Lanti, D., Rezk, M., Xiao, G., and Calvanese, D. (2015). The NPD benchmark: Reality check for OBDA systems. In *EDBT*, pages 617–628. OpenProceedings.org.

[20] Lanti, D., Xiao, G., and Calvanese, D. (2019). VIG: data scaling for OBDA benchmarks. *Semantic Web*, 10(2):413–433.

[21] Mark-Andor, S. (2026). Modern alternatives to hive: A systematic review and single-node benchmark of sql-on-hadoop and lakehouse engines. *Inf. Syst.*, 136:102635.

[22] Raasveldt, M. (2022). Duckdb - A modern modular and extensible database system. In *CDMS@VLDB*.

[23] Raasveldt, M. and Mühleisen, H. (2019). Duckdb: an embeddable analytical database. In *SIGMOD Conference*, pages 1981–1984. ACM.

[24] Van der Wal, D., Podareanu, D., Rodríguez, J. M., Silvello, G., and Romanovych, A. (2025). Deliverable 2.11: Federated infrastructure design. EU Project Deliverable D2.11, Hereditary Project.

[25] van der Walt, S., Colbert, S. C., and Varoquaux, G. (2011). The numpy array: A structure for efficient numerical computation. *Comput. Sci. Eng.*, 13(2):22–30.