TECHNISCHE UNIVERSITÄT BERLIN

A Web Service Backend for Archiving and Providing QDA Artifacts

Master's Thesis

Author
Tessa Heidkamp
353069
tessa.v.heidkamp@campus.tu-berlin.de

Examiners
Prof. Dr. Dirk Riehle
Prof. Dr. Manfred Hauswirth
Prof. Dr. Volker Markl

Technische Universität Berlin, 2026
Fakultät Elektrotechnik und Informatik
Fachgebiet Open Distributed Systems

A Web Service Backend for Archiving and Providing QDA Artifacts

Master's Thesis

Submitted by:
Tessa Heidkamp
353069
tessa.v.heidkamp@campus.tu-berlin.de

Technische Universität Berlin
Fakultät Elektrotechnik und Informatik
Fachgebiet Open Distributed Systems

2026

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne Hilfe Dritter und ausschließlich unter Verwendung der aufgeführten Quellen und Hilfsmittel angefertigt habe. Alle Stellen, die den benutzten Quellen und Hilfsmitteln wörtlich oder inhaltlich entnommen sind, habe ich als solche kenntlich gemacht.

Sofern KI-Tools verwendet wurden, habe ich Produktnamen, Hersteller, die jeweils verwendete Softwareversion und die Art der Nutzung (z. B. sprachliche Überprüfung und Verbesserung der Texte, systematische Recherche) benannt. Für die vorliegende Arbeit wurden folgende KI-Tools für die nachstehenden Zwecke eingesetzt:

- Claude Sonnet 4.6 (Anthropic): Rechtschreibung, Grammatik, stilistische Überarbeitung, sprachliche Überprüfung sowie Übersetzung einzelner Wörter und Formulierungen, außerdem visuelle Verbesserung von zuvor selbst entworfenen Grafiken, Unterstützung bei Programmieraufgaben, insbesondere zur Fehlersuche und Codeverbesserung und -überprüfung.
- ChatGPT GPT-5.5 (OpenAI): Rechtschreibung, Grammatik, stilistische Überarbeitung, sprachliche Überprüfung sowie Übersetzung einzelner Wörter und Formulierungen.

Ich verantworte die Auswahl, die Übernahme und sämtliche Ergebnisse des von mir verwendeten KI-generierten Outputs vollumfänglich selbst.

Die Satzung zur Sicherung guter wissenschaftlicher Praxis an der TU Berlin vom 30. Mai 2023 (https://www.static.tu.berlin/fileadmin/www/10002457/K3-AMBl/Amtsblatt_2023/Amtliches_Mitteilungsblatt_Nr._16_vom_30.05.2023.pdf) habe ich zur Kenntnis genommen.

Ich erkläre weiterhin, dass ich die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt habe.

(Unterschrift) Tessa Heidkamp, Berlin, 5. Juni 2026

Abstract

Qualitative data analysis tools such as ATLAS.ti, MAXQDA, and NVivo produce project archives that contain not only primary research material but also coding schemes, annotations, and interpretive structures. These emerge over often extended analytical processes. Existing research repositories largely ignore this internal structure. Instead, uploaded files are typically stored as opaque binary objects, leaving their analytical content invisible to the archive. Proprietary file formats further increase these limitations. Projects created with one qualitative data analysis tool cannot be reliably transferred to another, and the discontinuation of format support can compromise the long-term accessibility and usability of archived data. To address these limitations, the thesis designs and implements QDArchive, an archival system that manages qualitative research projects as structured, versioned, and interoperable research objects. Rather than treating them as static files, the architecture models these objects as domain entities with explicit lifecycle states, access semantics, and transformation histories. To support these requirements, QDArchive is implemented as a modular monolith. This architecture was chosen over a microservice architecture because the domain requires strong consistency across lifecycle transitions, authorization state, and publication visibility. In distributed systems, such guarantees can only be achieved at significant additional operational complexity. The internal structure of the system follows Domain-Driven Design principles and is organized into bounded contexts for archival management, format conversion, and the management of public access and discovery functions. Format conversion and interoperability between different file formats are handled through QDConvert, a dedicated conversion library developed alongside the system. This separation ensures that format-specific differences remain isolated from the system's archival and access-control mechanisms. The architecture is evaluated against established software quality attributes including maintainability, reliability, and interoperability. The thesis provides a domain-specific requirements analysis, a modular architectural design, and a working prototype that demonstrates the viability of preserving qualitative research artifacts as structured archival objects.

Kurzfassung

Software zur qualitativen Datenanalyse, wie beispielsweise ATLAS.ti, MAXQDA oder NVivo, erzeugt Projektarchive, die nicht nur primäres Forschungsmaterial enthalten, sondern auch Kodierschemata, Annotationen und interpretative Strukturen, welche im Verlauf oft langjähriger Analyseprozesse entstehen. Bestehende Forschungsrepositorien ignorieren diese interne Struktur weitgehend. Stattdessen werden hochgeladene Dateien meist als undurchsichtige Binärobjekte gespeichert, sodass ihr analytischer Inhalt dem Archiv verborgen bleibt. Proprietäre Dateiformate verschärfen diese Einschränkungen zusätzlich. Projekte, die mit einer bestimmten qualitativen Analysesoftware erstellt wurden, lassen sich nicht zuverlässig in eine andere übertragen. Wird die Unterstützung eines Formats eingestellt, kann dies die langfristige Zugänglichkeit und Nutzbarkeit der archivierten Daten gefährden. Um diesen Einschränkungen entgegenzuwirken, entwirft und implementiert diese Arbeit QDArchive, ein Archivsystem, das qualitative Forschungsprojekte als strukturierte, versionierte und interoperable Objekte verwaltet. Anstatt sie als statische Dateien zu behandeln, modelliert die Architektur Artefakte als Domänenobjekte mit expliziten Lifecycle-Zuständen, Zugriffssemantik und Transformationshistorien. Zur Umsetzung dieser Anforderungen wurde QDArchive als modularer Monolith konzipiert. Diese Architektur wurde einer Microservice-Architektur vorgezogen, da die Domäne starke Konsistenz über Lifecycle-Übergänge, Autorisierungszustand und Publikationssichtbarkeit erfordert. Solche Konsistenzgarantien lassen sich in verteilten Systemen nur mit erheblicher zusätzlicher Komplexität umsetzen. Die interne Struktur des Systems folgt den Prinzipien des Domain-Driven Designs und strukturiert das System in voneinander abgegrenzte Bounded Contexts für Archivverwaltung, Formatkonvertierung und die Verwaltung öffentlicher Zugriffs- und Suchfunktionen. Die Formatkonvertierung und Interoperabilität zwischen unterschiedlichen Dateiformaten werden durch QDConvert umgesetzt, eine dedizierte Konvertierungsbibliothek, die parallel zum System entwickelt wurde. Dadurch bleiben formatspezifische Unterschiede von den Archivierungs- und Zugriffsmechanismen des Systems getrennt. Die Architektur wird anhand etablierter Software-Qualitätsmerkmale evaluiert, darunter Wartbarkeit, Zuverlässigkeit und Interoperabilität. Die Arbeit liefert eine domänenspezifische Anforderungsanalyse, einen modularen Architekturentwurf und einen funktionierenden Prototyp für das Backend des Systems, das die Eignung des Ansatzes der Archivierung qualitativer Forschungsartefakte als strukturierte Archivierungsobjekte demonstriert.

Contents

1 Introduction **9**

2 Related Work and Background **12**
2.1 Scientific Archiving Systems . . . 12
2.2 Qualitative Data Repositories . . . 13
2.3 QDA Artifacts as Structured Research Objects . . . 15
2.4 Limitations of Current Systems . . . 16
2.5 Preservation and Access Control Requirements . . . 16
2.6 Implications for System Design . . . 17

3 Requirements **19**
3.1 System Scope and Context . . . 19
3.2 Stakeholders and User Roles . . . 20
3.3 Functional Requirements . . . 21
3.3.1 Find . . . 21
3.3.2 Convert . . . 21
3.3.3 Share . . . 22
3.3.4 Administrative Functions . . . 23
3.3.5 Background Services . . . 23
3.4 Non-Functional Requirements and Quality Attributes . . . 24
3.5 Quality Attribute Scenarios . . . 25
3.6 Requirements Summary . . . 28

4 Architecture of the QDArchive Backend **30**
4.1 QDArchive as a Modular Monolith . . . 30
4.2 Domain-Driven Design . . . 31
4.2.1 Subdomain Analysis . . . 31
4.2.2 Bounded Contexts . . . 32
4.2.3 Context Integration Patterns . . . 33
4.2.4 Ubiquitous Language . . . 34
4.3 Domain Model and Persistence Design . . . 34
4.3.1 Aggregate Structure . . . 35
4.3.2 Mapping to Persistence . . . 35
4.3.3 Separation of Archival Structure and Analytical Content . . . 36
4.4 Modularization . . . 37
4.4.1 Module Structure . . . 37
4.4.2 Module Responsibilities . . . 37
4.5 Application and Service Model . . . 37
4.5.1 Layered Responsibility Separation . . . 38
4.5.2 Dependency Composition . . . 38
4.6 Supporting Architectural Mechanisms . . . 39
4.6.1 Data Management . . . 39
4.6.2 Access Control Model . . . 39
4.6.3 Processing Model . . . 39
4.7 Architectural Trade-offs . . . 39
4.7.1 Infrastructure Dependency on Supabase . . . 40
4.7.2 Database-level versus Application-level Access Control . . . 40
4.7.3 Synchronous Request Processing . . . 41

4.7.4 Dependency Direction between Share and Convert 41
4.7.5 Compensating Transactions and Saga Coordination 41

5 Implementation 43
5.1 Technology Stack 43
5.2 System Structure 44
5.2.1 Module Mapping 45
5.2.2 Layered Separation 46
5.3 Data Model and Persistence 48
5.3.1 Core Tables 49
5.3.2 Indexing and Search 51
5.3.3 Integrity Enforcement 51
5.4 Access Control 52
5.4.1 Role-Based Access 53
5.4.2 State-Aware Access 54
5.5 Application Logic and API Design 56
5.5.1 Resource Structure 56
5.5.2 Route Handler Conventions 57
5.5.3 Request Lifecycle 57
5.6 Integration of External Services 58
5.6.1 QDConvert 58
5.6.2 Infrastructure 60
5.7 Deployment 61

6 Evaluation 63
6.1 Evaluation Methodology 63
6.2 Requirement Coverage Analysis 64
6.3 Scenario Validation 67
6.3.1 Modifiability Scenario (Adding a New QDA Format) 68
6.3.2 Security Scenario: Share Link Enumeration 68
6.3.3 Security Scenario: Privilege Isolation (Application-Layer Bug Bypasses Ownership Check) 68
6.3.4 Access Control Consistency Scenario (Share Link Invalidation) . . . 68
6.3.5 Reliability Scenario 69
6.3.6 Anonymous Session Upgrade Scenario 69
6.4 Modularity and Coupling Analysis 69
6.4.1 Cohesion Within Modules 69
6.4.2 Realization of the Aggregate Model 70
6.4.3 Inter-Module Communication 71
6.5 Scalability and Performance Considerations 71
6.5.1 Vertical Scaling and Concurrency 71
6.5.2 Database Access Patterns 72
6.5.3 Memory-Intensive File Processing 72
6.5.4 Horizontal Scaling 73
6.6 Security Analysis 73
6.6.1 Authorization Enforcement Points 73
6.6.2 Share Link Security 74
6.6.3 Potential Attack Vectors 74
6.6.4 Guest Session Security 75
6.6.5 Trust Assumptions 75

6.7 Maintainability and Extensibility 75
6.7.1 Format Extension 76
6.7.2 Publication and Access-Control Extension 76
6.7.3 Schema Evolution 76
6.8 Overall Assessment 76

7 Next Steps **79**
7.1 Closing the Gap Between Model and Runtime 79
7.2 Operational Extensions 80
7.3 Potential Architectural Developments 80

8 Conclusion **83**
8.1 Summary of Contributions 83
8.2 Architectural Reflection 83

References **85**

List of Figures

2.1 Representative repository systems positioned conceptually by methodological focus and openness of access 14
2.2 Conceptual structure of a QDA artifact as an interconnected analytical object 15
3.3 System context of QDArchive and its external actors and service dependencies 19
4.4 High-level modular structure of the QDArchive backend 30
4.5 Bounded contexts and subdomain classification of QDArchive 33
5.6 Layered request flow and dependency composition throughout the QDArchive backend 47
5.7 Entity-Relationship Diagram of the core database schema 49
5.8 Factors determining effective Project visibility and access in QDArchive . . 52
5.9 Request lifecycle from client to database interaction 57
5.10 Conversion pipeline from file upload to export and persistence 59
5.11 Mapping of conversion output to domain entities 60

List of Tables

3.1 Mapping of functional requirements to solution areas 28
5.2 Mapping of domain concepts to implementation modules 45
6.3 Aggregated mapping of functional requirements (Section 3.3) to architectural mechanisms and implementation status 65
6.4 Mapping of quality attributes (Section 3.4) to architectural mechanisms and implementation status 66

1 Introduction

The scientific research community increasingly relies on digital systems to collect, organize, preserve, and interpret empirical material. In qualitative research in particular, computer-assisted qualitative data analysis (CAQDAS) tools such as ATLAS.ti, MAXQDA, and NVivo have become central components of the analytical workflow (Chandra & Shang, 2019). These tools produce research artifacts in the form of structured project archives that combine primary research material, such as interview transcripts, images, audio recordings or video files, with layers of analytical interpretation including codes, code hierarchies, annotations and analytical memos. Although these artifacts are often treated operationally as ordinary project files, they are in reality complex and evolving research objects that encapsulate the analytical process itself. At the same time, the scientific community has increasingly accepted that research findings should not only be published but also remain verifiable and accessible for future analysis (OECD, 2007; UNESCO, 2021).

One important driver behind these developments has been the reproducibility crisis across empirical disciplines, where multiple studies and surveys have shown that published findings are often difficult or impossible to reproduce because the underlying data and analytical decisions are insufficiently documented or unavailable (Baker, 2016; Ioannidis, 2005; Open Science Collaboration, 2015).

These concerns are especially relevant in qualitative research, where analytical interpretation is closely tied to the underlying research material and often difficult to reconstruct from publications alone. The systematic archiving of qualitative datasets enables secondary analysis, long-term preservation, and cross-study validation (Corti, 2007). However, qualitative research artifacts differ fundamentally from conventional quantitative dataset. Whereas quantitative research data is typically represented in relatively portable tabular structures, CAQDAS projects are deeply entangled with proprietary software ecosystems and exhibit substantial structural heterogenity. E.g., a project created in ATLAS.ti cannot reliably be opened in NVivo or MAXQDA without information loss, and archived project files may become inaccessible when proprietary formats evolve incompatibly or supporting software is discontinued. This heterogeneity is therefore not merely an inconvenience but a long-term preservation risk. As a result, there looms a threat that qualitative projects (containing years of analytical work) may become partially or entirely unreadable within a relatively short time period if no migration or interoperability strategy exists (Corti & Gregory, 2011).

These preservation challenges cannot be addressed solely at the level of individual CAQDAS tools. Long-term accessibility and reuse instead depend on scientific archiving infrastructure that is capable of preserving not only files themselves but also the analytical structures embedded within them. However, existing general-purpose research repositories are primarily designed for depositing tabular datasets, documents, or static media files and therefore largely treat uploaded content as opaque binary objects (Corti & Gregory, 2011; Karcher et al., 2016). Consequently, they cannot meaningfully inspect, preserve, or manage the internal analytical structure of Qualitative Data Analysis (QDA) artifacts, including coding systems, document relationships, interpretive annotations, and workflow semantics. Much of what gives qualitative projects their scientific value therefore remains effectively invisible to the archive itself and inaccessible to future researchers interacting with the deposited material.

In summary, these limitations reveal that QDA artifacts cannot be treated as passive archival files. Instead, they require domain-aware lifecycle management, structured interoperability handling across heterogeneous CAQDAS formats, and fine-grained publication control as first-class architectural concerns (CCSDS, 2024; Evers et al., 2020). Conventional file-oriented archival systems fail to capture the structural and semantic properties inherent in QDA artifacts and therefore cannot adequately support artifact-specific lifecycle transitions, controlled sharing workflows, or long-term preservation strategies.

The central architectural problem addressed in this thesis is therefore the following: how can a backend system model QDA artifacts not as static binary files but as structured domain entities with their own lifecycles, semantics, interoperability requirements, and access rules? The short answer is that managing QDA artifacts as first-class domain entities introduces architectural challenges that extend beyond conventional file storage systems. Artifact lifecycle transitions affect visibility, authorization, and accessibility simultaneously, requiring coordinated consistency across multiple entities and system layers. At the same time, authorization is inherently multi-dimensional. Access decisions depend not only on user identity, but also on publication state, sharing configuration, and temporal constraints. In addition, the system must integrate external conversion services for proprietary QDA formats without coupling the core domain model to externally maintained libraries or file representations.

These requirements shape the architecture more strongly than technological preference alone. This thesis argues that a modular monolithic architecture is particularly suitable for the domain-specific requirements of QDA artifact preservation and publication workflows. A fully distributed microservice architecture would introduce coordination overhead and transactional complexity that are difficult to justify for tightly coupled archival workflows (Fowler, 2015; Newman, 2021), while a conventional monolith risks entangling lifecycle management, authorization, conversion, and persistence concerns into a rigid codebase (Richards & Ford, 2020). A modular monolith therefore represents a compromise between strong consistency and explicit architectural separation.

Against this background, the thesis examines the following research questions:

1. How can analytical structures in heterogeneous CAQDAS formats be preserved through a format-agnostic archival architecture?
2. How can lifecycle-aware access control be integrated into the domain model without compromising architectural modularity?
3. To what extent does the resulting architecture preserve domain consistency while supporting extensibility, maintainability, and architectural evolution?

To address these questions, the thesis first defines a set of functional and non-functional requirements for the web-based QDA artifact archiving backend based on domain analysis and digital preservation principles. Subsequently it proposes a modular monolithic backend architecture that models artifact lifecycles, publication workflows, interoperability concerns, and authorization semantics as explicit architectural concerns. Finally, the thesis presents an implemented prototype together with a systematic evaluation against established software quality attributes and architectural goals.

This thesis is structured into five parts. Chapter 2 and Chapter 3 establish the conceptual and methodological foundation of the work. The related work chapter reviews existing scientific archiving systems, qualitative data repositories, and preservation challenges asso-

ciated with QDA artifacts. Building on this foundation, the requirements chapter defines the functional and non-functional requirements as well as the quality attribute scenarios that shape the architectural design. Chapter 4 and Chapter 5 present the proposed system itself. The architecture chapter introduces the modular decomposition, the domain model, and the mechanisms for lifecycle management and access control. The implementation chapter then describes how these architectural concepts are realized in practice, including the persistence model, API structure, and deployment strategy. Chapter 6 evaluates the architecture against the previously defined quality goals and architectural scenarios, while Chapter 7 discusses limitations, future extensions, and possible architectural evolutions. Finally, Chapter 8 summarizes the results of the thesis and reflects on broader implications for qualitative research preservation systems.

2 Related Work and Background

This chapter reviews existing approaches to scientific archiving and qualitative data management while laying the conceptual foundation for QDArchive, the archival system developed in this thesis. It begins by analyzing general-purpose research archives, then examines archiving systems specifically designed to handle qualitative research data. From there, the chapter looks more closely at the particular structure of Qualitative Data Analysis (QDA) artifacts, and the particular demands they place on long-term preservation and access control. Finally, it draws out a set of design implications that shape the system architecture developed in later parts of the thesis.

2.1 Scientific Archiving Systems

Scientific archiving systems have continuously developed in order to increase the possibilities to preserve, share and use research data (Corti, 2007). In more recent research contexts, these efforts are increasingly associated with broader demands for transparency, reproducibility, and research validation (Baker, 2016; OECD, 2007; UNESCO, 2021). To support these objectives, a number of infrastructures have been developed. General-purpose multidisciplinary repositories such as Zenodo, Figshare, and the Harvard Dataverse enable researchers to deposit and disseminate datasets in a standardized and accessible manner (CERN, n.d.; Figshare, n.d.; Harvard Dataverse, n.d.). Alongside these platforms, institutional repositories and domain-specific archives, such as the UK Data Archive, provide more specialized support tailored to particular research communities (UK Data Service, n.d.). Together, these systems form a distributed ecosystem for the long-term preservation and controlled sharing of research data (Wilkinson et al., 2016). Some platforms, however, go beyond the role of such a repository. The Open Science Framework (OSF), for example, integrates data archiving with project management and pre-registration, thereby supporting multiple stages of the research process within a single environment (Foster & Deardorff, 2017). In the European context, infrastructures such as the Consortium of European Social Science Data Archives (CESSDA, 2017) and the Dutch Data Archiving and Networked Services (DANS, 2024) provide national and cross-border support for long-term data archiving services and sustainable data stewardship across multiple disciplines.

Across these research data repositories, several recurring platform features have become common practice (Wilkinson et al., 2016). These platforms typically offer core services including persistent identifiers (mostly DOIs), version control, and embargo options that align data release with publication timelines. In addition, metadata standards like the Data Documentation Initiative (DDI, 2024) help document the methodological context in which data were generated.

While these features significantly enhance the discoverability and citation of research data, they primarily operate at the level of external description (Faniel et al., 2013). That is, they provide structured information about datasets without necessarily capturing their internal organization. As a result, the internal structure, hierarchies and analytical layers within the data often remain implicit. This approach works reasonably well for quantitative or relatively simple file-based data collections. For more complex research outputs, especially those that embed the full analytical process of qualitative research, it falls short.

In such cases, meaning is not contained in individual data points alone, but emerges from the relationships between them and from the analytical processes applied during the study.

Once detached from their original context, such data can be difficult to interpret and reuse without substantial domain knowledge (Borgman, 2012). Secondary users are frequently required to reconstruct the underlying structure and analytical logic manually, as these are rarely made explicit within existing archiving systems. As a consequence, general-purpose repositories are comparatively considered effective at storing and disseminating data, but they are less well suited for preserving artifacts in which meaning depends on interconnected elements and interpretive frameworks. This limitation becomes particularly evident when archiving qualitative research materials (Karcher et al., 2016).

2.2 Qualitative Data Repositories

In Addition to the general-purpose scientific archiving systems, a number of repositories have been developed specifically for depositing qualitative research data. These infrastructures build on existing general archiving approaches whilst addressing specific methodological and ethical requirements of qualitative research. Such requirements include, for example, the need for careful anonymization, controlled access to sensitive materials, and the management of informed consent. Typical data types include interview transcripts, field notes, audiovisual recordings, and observational data (Chandra & Shang, 2019).

Prominent examples of such repositories include the UK Data Service (UK Data Service, n.d.), the Inter-university Consortium for Political and Social Research (ICPSR, n.d.), and the Qualitative Data Repository (QDR) at Syracuse University (Syracuse University, n.d.). Compared to general-purpose systems, these platforms put a greater emphasis on contextual documentation and access control. Often, researchers are supported in preparing data for reuse by providing guidance and infrastructure for anonymization, consent management, and controlled-access workflows (Karcher et al., 2016). At the same time, these additional safeguards and domain-specific workflows can limit openness and interoperability compared to more generic repository infrastructures. ICPSR, for instance, has established the Qualitative Data Sharing (QDS) initiative, which curates collections specifically intended for methodological research and secondary analysis (Inter-university Consortium for Political and Social Research (ICPSR), n.d.). Existing repository infrastructures differ not only in their disciplinary focus, but also in the degree to which they prioritize open accessibility versus controlled access mechanisms for sensitive research data. This creates a spectrum between broadly accessible general-purpose repositories and more specialized qualitative archives with stricter governance and access-control requirements, as illustrated in Figure 2.1.

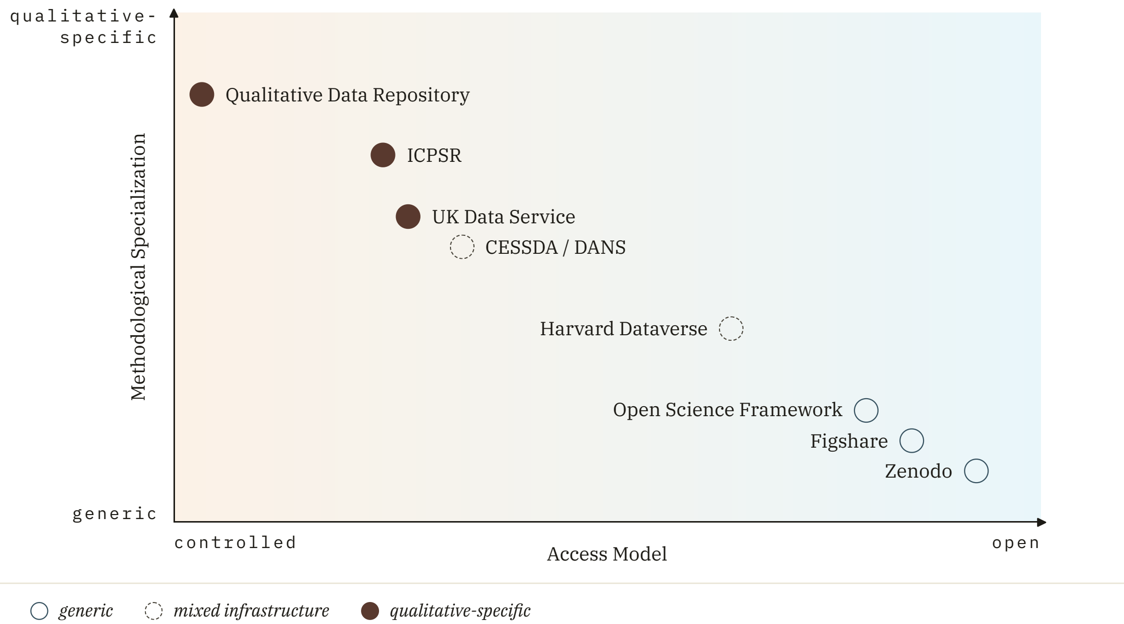

Figure 2.1: Representative repository systems positioned conceptually by methodological focus and openness of access

Despite their stronger focus on qualitative research practices, most of these existing repositories continue to focus primarily on raw or lightly processed data (Bishop & Kuula-Luumi, 2017; Karcher et al., 2016). They typically store qualitative data in formats such as text documents or transcripts. The analytical structures generated during analysis (coding schemes, annotations, memos, and code–data linkages) are rarely preserved in a machine-readable, interoperable form. When researchers deposit project files generated during the qualitative analysis process, they are often treated as black-box archives or exported to generic formats that discard relational information. As a result, critical components of qualitative research are often lost during archiving (Corti & Gregory, 2011; Evers et al., 2020).

Ethical considerations further complicate the situation. Qualitative data often contain information that cannot be fully anonymized without altering its interpretive value (Bishop, 2009). This creates a need for more fine-grained access control mechanisms, such as embargoes, user-specific permissions, and tiered access models, rather than simple public or private classifications. A comparative analysis of repository policies across ICPSR, QDR, the UK Data Service, and other platforms reveals considerable variation in requirements for participant consent documentation, acceptable file formats, and de-identification practices (Antes et al., 2017). This variation suggests that a widely accepted set of best practices has yet to emerge.

Recent initiatives have begun to address some of these limitations. The QualiCO ontology, e.g., provides a formal vocabulary for representing qualitative coding structures, thereby supporting the reuse of analytical frameworks (Hocker et al., 2021). In a similar spirit, workshops on Computer-assisted qualitative data analysis software (CAQDAS) tools and digital repositories have produced recommendations for making analyzed projects more shareable and transparent (Karcher et al., 2016). A persistent gap remains between the richness of qualitative analysis as conducted in research tools and what is ultimately

preserved in archival systems, as these approaches have so far seen limited adoption in operational repository infrastructures (Karcher et al., 2016).

2.3 QDA Artifacts as Structured Research Objects

CAQDAS tools such as ATLAS.ti, NVivo, and MAXQDA have significantly shaped contemporary qualitative research practices (Chandra & Shang, 2019). These tools support systematic coding, memoing, and visualization of complex datasets, enabling researchers to work with heterogeneous materials in a structured and iterative manner. The project files these tools produce (QDA artifacts) combine primary data (e.g. transcripts, images, audio or video) with multiple layers of secondary analysis including hierarchical code systems, coded segments, links, and interpretive memos. In contrast to conventional datasets, these artifacts capture not only the underlying data but also parts of the analytical process applied to it. They can therefore be understood as structured research objects that embed both data and interpretation.

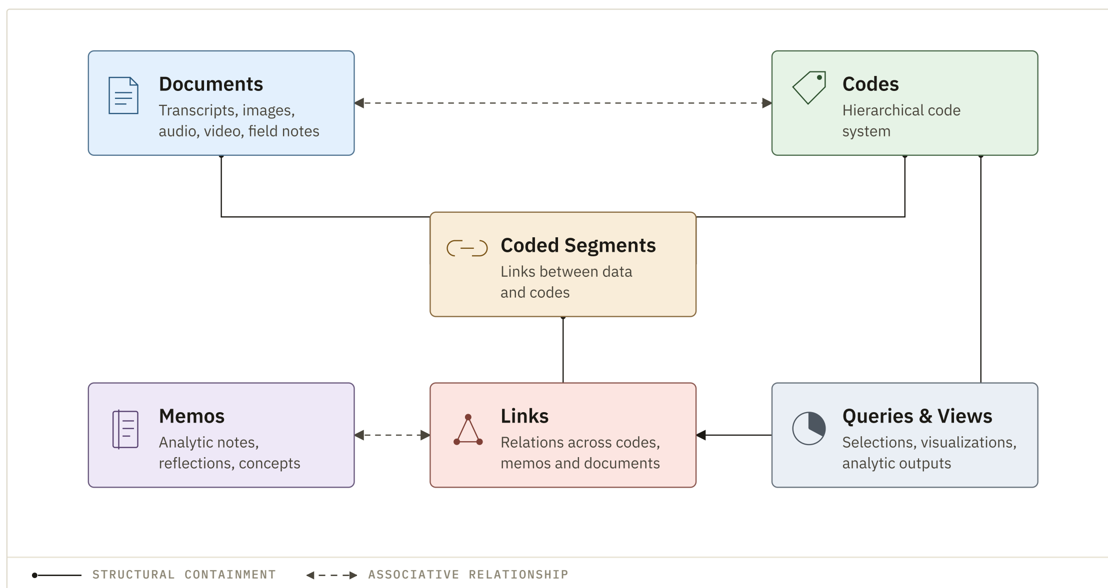

Figure 2.2: Conceptual structure of a QDA artifact as an interconnected analytical object

As illustrated in Figure 2.2, the meaning of a QDA artifact does not reside in individual files but emerges from the network of relationships connecting documents, codes, memos, and analytical outputs. QDA projects therefore resemble interconnected research structures rather than flat collections of files. This relational character creates important preservation challenges. Archival systems that treat project artifacts merely as opaque binary objects risk preserving the files themselves while losing the analytical context that gives them interpretive meaning.

This interconnected structure gives QDA artifacts a strong graph-like character. Nodes represent data segments, codes, or memos, while edges capture relationships such as code hierarchies, co-occurrences, or explicit links. These structures are not static. As analysis progresses, codes are refined, merged, or reorganized, and new connections between elements are continuously established (Saldaña, 2021). QDA artifacts typically combine multiple data modalities. Textual data such as transcripts are often linked to audio recordings,

videos, images, or field notes, allowing researchers to move between different representations of the same material. Additional analytical elements, such as annotations, memos, and layered coding structures, further increase the complexity of these interconnections. Much of the interpretive power of qualitative analysis lies precisely in these relationships (Mason, 2002). Preserving these interconnected structures in an archival system presents a representational challenge. Conventional file-based storage models capture files but not the relationships between them, while fully relational approaches require the archive to understand and model every format-specific relationship type (CCSDS, 2024; Corti & Gregory, 2011; Evers et al., 2020). A practical middle ground is to preserve the analytical structure as a format-agnostic serialized representation, separating the archival concerns of ownership, versioning, and access control from the internal structure of the artifact itself.

2.4 Limitations of Current Systems

Despite the structured nature of QDA artifacts, current software and archival systems provide only limited support for preserving their full analytical richness. One major limitation concerns the handling of analytical evolution. Qualitative analysis is inherently iterative and non-linear. Coding schemes change over time, categories are refined or reorganized, and interpretations are revisited (Saldaña, 2021). Yet most CAQDAS tools offer only limited versioning capabilities (Corti & Gregory, 2011; Evers, 2018). Changes are often overwritten or stored in ways that make the development of the analysis difficult to reconstruct. As a result, earlier states, intermediate interpretations, and discarded analytical paths are frequently lost or remain implicit.

A second limitation relates to format heterogeneity and interoperability. Each major CAQDAS package uses proprietary data formats, resulting in strong vendor lock-in. These formats are typically incompatible with one another, and exporting projects between tools often leads to significant information loss, particularly for complex features such as multimedia linkages, queries, or relational structures (Corti & Gregory, 2011). The REFI-QDA standard represents an important step toward addressing this issue. Developed as an open XML-based exchange format, it supports the transfer of code systems and complete projects across different tools (Evers, 2018; Evers et al., 2020). While adoption has increased and major software packages now offer some level of support, the standard does not fully capture all analytical features and can still lead to data loss in practice. Moreover, its integration into archival infrastructures remains limited (Evers et al., 2020; Karcher et al., 2016).

Together, these limitations create significant preservation risks. As software evolves or becomes obsolete, QDA artifacts may become partially or entirely inaccessible. More importantly, even when data remain available, the analytical structures that give them meaning may not survive in a usable form.

2.5 Preservation and Access Control Requirements

The long-term preservation of QDA artifacts therefore requires more than simple data storage. The Open Archival Information System (OAIS) reference model provides a widely used conceptual framework for digital preservation, emphasizing that both content and its interpretability must be maintained over time (CCSDS, 2024). Originally published as an ISO standard in 2003 and updated in subsequent revisions, the model has become a de facto reference for the design of digital archives across scientific disciplines. In this con-

text, interpretability includes not only metadata, but also the relationships and analytical structures embedded within the data.

At the same time, the FAIR principles (findability, accessibility, interoperability, and reusability) have become a widely accepted benchmark for research data management (Wilkinson et al., 2016). While these principles are increasingly applied in quantitative domains, their implementation in qualitative research is more complex. Ethical considerations, the importance of contextual richness, and the sensitivity of data often stand in tension with full openness (Antes et al., 2017; Bishop, 2009). Qualitative data frequently contain information that cannot be fully anonymized without altering its interpretive value. As a result, repositories must support more nuanced access control mechanisms, including embargoes, role-based permissions, and controlled access environments. In practice, qualitative research projects typically move through different lifecycle stages (private, shared, and publicly accessible) which require flexible yet robust management of access rights.

Existing repository systems provide some of these capabilities, but rarely in an integrated and researcher-friendly manner. In particular, they often lack support for combining structured representation of analytical content with fine-grained lifecycle management. This gap highlights the need for systems that can simultaneously address preservation, interpretability, and controlled accessibility.

2.6 Implications for System Design

The preceding analysis highlights several persistent gaps in existing infrastructures for qualitative data archiving. These gaps have direct implications for the design of systems intended to support the preservation and reuse of QDA artifacts.

First, current repositories provide insufficient support for structured QDA artifacts. Analytical layers, such as coding schemes, memos, and relational links, are often treated as secondary or remain implicit. A suitable system must therefore preserve and expose the full network of relationships between primary data and analytical elements. Second, interoperability remains limited. Proprietary formats and only partial support for standards such as REFI-QDA lead to vendor lock-in and potential information loss. To address this issue, systems require a vendor-neutral internal representation capable of ingesting heterogeneous formats and supporting standardized export. Third, existing infrastructures offer only limited support for lifecycle management and fine-grained access control. Qualitative research workflows require smooth transitions between private, shared, and publicly accessible states, while maintaining auditability and respecting ethical constraints. This calls for integrated mechanisms that combine access control with traceability. Finally, search and discovery capabilities for qualitative data remain underdeveloped (Faniel & Jacobsen, 2013; Tenopir et al., 2011). While file-level indexing is common, structured search across codes, memos, and analytical relationships is rarely supported. As a result, important aspects of qualitative projects remain difficult to explore and reuse.

From a design-science perspective (Hevner et al., 2004), QDArchive is conceived as a research artifact that addresses these shortcomings through the design and implementation of a dedicated software system. It provides automated format detection and conversion, a unified internal representation of QDA artifacts, lifecycle-aware access control (private, shared, and public states with licensing), persistent identifiers, and public search capabilities.

These requirements translate into several key architectural priorities. The system must include a flexible import and export layer that supports both proprietary CAQDAS formats and standards such as REFI-QDA. It requires a relational internal model capable of representing hierarchical codes, linkages, and multimodal data. In addition, robust access-control and state-transition mechanisms are needed to support core user workflows such as finding, converting, and sharing data. Finally, preservation-oriented features including versioning, metadata enrichment, and OAIS-informed packaging are essential to ensure long-term usability.

Taken together, these challenges reveal that qualitative data archiving cannot be treated merely as a file-storage problem. Supporting structured QDA artifacts requires architectural mechanisms for interoperability, lifecycle-aware access control, preservation-oriented data management, and meaningful discovery across analytical structures. The identified gaps therefore translate directly into functional and non-functional design requirements that shape the system architecture presented in the following chapters. Chapter 3 derives these requirements systematically and discusses the architectural pressures and trade-offs that emerge from balancing openness, controlled access, interoperability, and long-term maintainability.

3 Requirements

Based on the analysis presented in Chapter 2, this chapter establishes the requirements for QDArchive. The requirements were shaped by three competing concerns that emerged from the related work: (i) openness for anonymous and low-commitment users, (ii) strict access control for private and selectively shared research data, and (iii) long-term format interoperability in a ecosystem where proprietary CAQDAS formats remain the norm. Balancing these concerns led to several non-obvious architectural decisions.

Functional requirements describe the externally observable behavior of the system, whereas non-functional requirements capture quality attributes that shape the system architecture. To improve clarity and support systematic evaluation, important quality requirements are specified using quality attribute scenarios (Bass et al., 2021).

3.1 System Scope and Context

QDArchive is a web-based service for managing qualitative research artifacts. Its responsibilities include artifact storage, versioning, format conversion, and controlled sharing. The service depends on four external infrastructure services for (i) authentication, (ii) relational persistence, (iii) binary object storage, and (iv) format conversion. The application layer orchestrates these services but does not implement their core functionality. The architectural focus of this thesis therefore lies on how the application itself implements the logic that governs project ownership, visibility transitions, and artifact sharing, as well as how these external services are integrated into the backend domain model and application structure. Figure 3.3 illustrates the external actors and infrastructure dependencies that interact with the system.

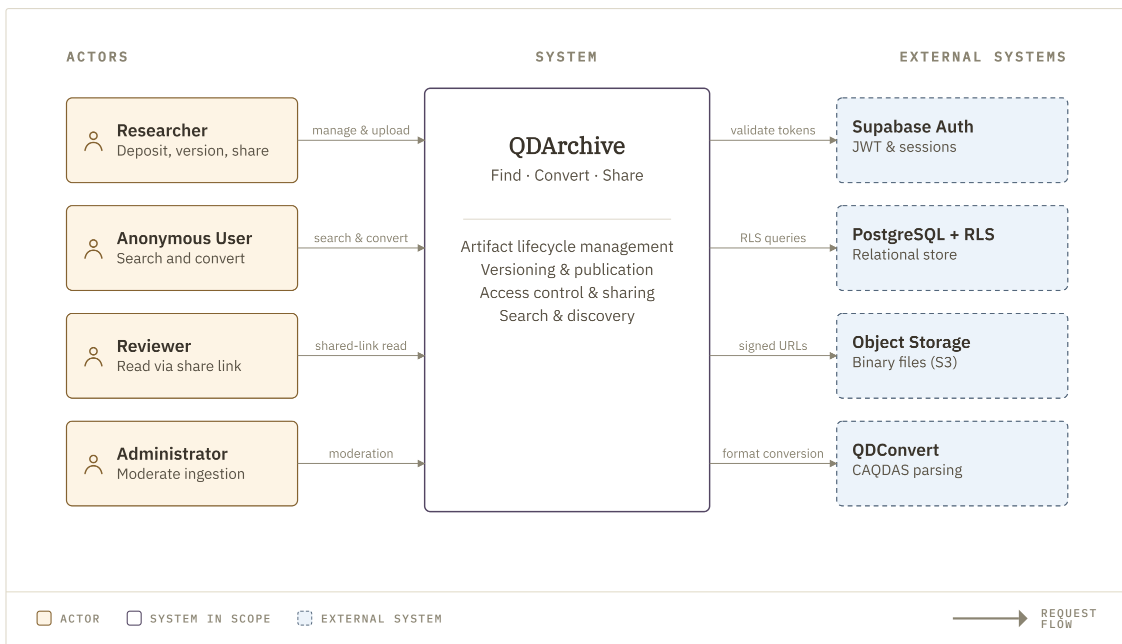

Figure 3.3: System context of QDArchive and its external actors and service dependencies

The system coordinates three primary functional areas. Find provides public artifact discovery and full-text search. Convert transforms QDA artifacts between supported

CAQDAS formats and is available to both guest and registered users. Share manages artifact ownership, versioning, publication, and access control and therefore requires an authenticated identity.

QDArchive supports both authenticated and anonymous interaction modes. In authenticated mode, a verified user identity governs all ownership and access control decisions. In anonymous mode, temporary guest sessions allow users to search public artifacts and convert files without prior registration while still associating created artifacts with a stable identifier. Anonymous sessions can later be associated with a registered account while preserving previously created artifacts.

Access control is enforced at two independent layers. The application layer performs ownership and permission checks before executing domain operations, while row-level security policies at the database tier enforce visibility constraints independently of application behavior (PostgreSQL, 2026f). This layered model follows directly from the coexistence of private, shared, and publicly accessible artifacts within the same system.

The architecture assumes that uploaded CAQDAS files conform to their respective format specifications. Malformed input is detected and rejected at the conversion boundary but is not corrected automatically. External infrastructure services are assumed to be available and reachable. The system does not implement fallback behavior for infrastructure outages. Authentication tokens are assumed to originate from a trusted GoTrue instance, with token integrity enforced through operational secret management rather than application-level cryptographic verification.

QDArchive is operated by a small development team in a self-hosted environment. The system is deployed as a single containerized unit using Docker Compose, constraining the current deployment model primarily to vertical scaling. All format-specific parsing and serialization logic resides in the external QDConvert library. The application layer does not interpret CAQDAS formats directly. Administrative database access is intentionally excluded from the system design, meaning that all data operations are mediated through the application API.

UI component implementation and client-side rendering logic are excluded because the thesis focuses on backend architecture and its quality properties. Interpretation of proprietary CAQDAS formats is delegated entirely to the external conversion library and therefore falls outside the architectural responsibility of the application layer. Infrastructure provisioning, deployment pipeline configuration, and database administration are treated as operational concerns rather than core architectural design problems. Similarly, background crawling, automated ingestion, and scheduled backup services are defined in the broader system specification but require additional scheduler infrastructure and have therefore been deferred from the current implementation scope.

3.2 Stakeholders and User Roles

Users who interact with the system are researchers, anonymous users, reviewers, and administrators, operating under different trust levels reflected in the access control design. Their roles are described in the following section. Researchers are the primary users. They produce qualitative research artifacts using CAQDAS tools and need to deposit, version, and share these artifacts with configurable access levels. Researchers may also discover and download artifacts published by others.

Anonymous users interact with the system without a registered account through two primary pathways. They may search for and download publicly available artifacts, or they may convert QDA files between supported formats as guests. In both cases, acceptance of the Terms of Service is required before the respective functionality becomes available. Guest-created conversion artifacts files remain temporarily accessible during the active session and can later be transferred seamlessly if the user decides to register an account.

Reviewers are registered users who have been granted read access to a private research projects by its owner through an explicit invitation. They can view the research project after logging in, but it remains excluded from public search results. Reviewer access is modeled as a permission level on the share-link entity.

Administrators have unrestricted read and write access to all system data for operational and moderation purposes. In the intended system design they operate a review queue for externally sourced projects.

3.3 Functional Requirements

The functional requirements are organized around three user-facing solution areas that reflect the core value proposition identified in the related work: (i) Find addresses the discovery and accessibility of archived qualitative research data, (ii) Convert addresses the format interoperability gap that currently prevents researchers from reusing data across CAQDAS tools, and (iii) Share addresses the lifecycle management and controlled publication of research artifacts. This structure also maps directly onto the trust model of the system. Find and Convert are available without registration to minimize barriers to access, while Share requires an authenticated identity because it involves persistent ownership and access control obligations.

3.3.1 Find

F1 The system shall provide a public search interface accessible without authentication.

F2 The search interface shall support full-text search across research project names and descriptions. Results shall be presented as a paginated list of matching research projects.

F3 Only research projects that have been explicitly set to public shall appear in search results.

F4 The search interface shall support filtering by one or more tags, with a configurable tag-matching mode (any/all). Multiple tags may be applied simultaneously.

F5 The search interface shall support filtering by author and by creation or modification date ranges.

F6 Search results shall be sortable by creation date and by last modification date, in ascending or descending order.

3.3.2 Convert

F7 The system shall provide a format conversion interface for guest and registered users.

F8 Unauthenticated users shall be required to accept the Terms of Service before the conversion function is enabled.

F9 The system shall automatically identify the format of an uploaded QDA file based on its structure. Unsupported or malformed files shall be rejected with a descriptive error message indicating the reason for rejection.

F10 The system shall expose the set of supported target formats dynamically, derived from the capabilities of the installed conversion library.

F11 Users shall be able to select the target format and, upon successful conversion, download the converted file.

F12 For each conversion, the system shall create a private research project associated with the current user session (anonymous or authenticated). The uploaded source file and the converted output shall be stored as versioned files within this project.

F13 Multiple conversions performed within the same anonymous session shall be associated with the same anonymous user identifier, making them retrievable for the duration of the session as recent converted files.

F14 A guest user who has performed a conversion shall be able to create a registered account immediately after conversion without losing access to the converted artifact. The session upgrade shall preserve the anonymous user's project ownership.

F15 Authenticated users shall be able to view and edit the projects created from conversions in the "My Projects" section.

F16 If an uploaded artifact contains references to primary data files that are not embedded in the archive, the system shall identify and display these missing references. Users shall be offered the option to supply the missing files, but this shall not be required to proceed with the conversion.

3.3.3 Share

Account management

F17 The system shall support user registration with email and password. Upon registration, users shall be required to accept the Terms of Service and shall provide a profile with at minimum a given name, family name.

F18 The system shall support login with email and password.

F19 The system shall support logout, which terminates the authenticated session.

F20 The system shall support password recovery via a time-limited link sent to the user's registered email address.

F21 Authenticated users shall be able to update their profile information including name, title, city, country, and avatar image.

Archive and Versioning

F22 Authenticated users shall be able to create research projects. Each project shall carry a name, an optional description, zero or more tags, and a visibility state. The default visibility state at creation shall be private.

F23 Users shall be able to create new versions of an existing project. When creating a new project version, the user may optionally copy files from the previous project

version. Each project version shall record a project version number, an optional project version name, and an optional description.

F24 Users shall be able to upload one or more files to a project version. Each file shall carry a name, an optional description, and a declared file type (primary data, supplementary, or converted output).

F25 Users shall be able to update file metadata (name and description) after upload.

F26 Users shall be able to delete individual files from a project version, individual project versions from a project, or an entire project including all project versions and files.

F27 Users shall be able to download an entire project version as a single archive.

Sharing and Publication

F28 Users shall be able to generate a secure share link for a private project. The link shall make the project world-readable without requiring authentication and without making the project publicly discoverable through search. The link shall be stable until the user explicitly regenerates or deletes it, and shall be independently disableable without regeneration.

F29 Users shall be able to regenerate a share link, which shall immediately invalidate the previous link and issue a new one.

F30 Users shall be able to set a project to public, making it discoverable through the Find interface and accessible via a stable public link. Setting a project to public shall invalidate any previously issued share link.

3.3.4 Administrative Functions

F31 Administrators shall have access to a review queue containing entries for externally sourced qualitative research projects. Each entry shall consist of a link to the external project.

F32 Administrators shall be able to approve or reject entries in the review queue. Approved Projects shall be ingested into the archive. Rejected entries shall be removed from the queue.

3.3.5 Background Services

F33 The system shall periodically scan a defined set of external repositories using preconfigured search queries. Newly discovered qualitative research projects shall be added to the administrator review queue.

F34 The system shall periodically check externally sourced projects that have already been ingested for changes. Projects for which a change is detected shall be submitted to the administrator review queue.

F35 The system shall perform regular backups of all stored data to secondary storage.

Background services are defined in the system specification but are outside the scope of the current implementation, which focuses on the interactive web backend and its architecture.

3.4 Non-Functional Requirements and Quality Attributes

The non-functional requirements are expressed as quality attributes following the ISO/IEC 25010 product quality model (ISO/IEC, 2023). The attributes selected reflect the architectural challenges identified in the related work and the constraints of the deployment context.

Maintainability: The system shall be structured such that individual modules can be modified or replaced without requiring changes to unrelated parts of the codebase. The integration with the external conversion library shall be encapsulated behind a dedicated adapter (Richards & Ford, 2020) so that support for new QDA formats can be added by modifying a single class without affecting the artifact lifecycle or access control subsystems. This directly addresses the interoperability gap identified in the related work, where format lock-in is a primary risk to long-term preservation.

Security: The system shall enforce access control independently at both the database and the application layer, so that no application-level bug alone can result in unauthorized data access. Authentication is based on JSON Web Tokens (JWTs) (Jones et al., 2015), which are issued and validated by the authentication service. Secure share link tokens shall provide sufficient entropy (OWASP Foundation, 2024) to make enumeration attacks infeasible. Credentials with administrative database privileges shall never be exposed to client-side code. All privileged database operations shall be executed through server-side API routes using a service-role key that is kept strictly server-side. Anonymous sessions shall be isolated from one another and shall not be able to access each other's projects.

Scalability: Two characteristics of QDArchive make scalability a relevant concern. First, format conversion is a CPU-bound operation. Parsing and re-serializing a large CAQDAS project file can be computationally intensive, and multiple concurrent conversion requests must not degrade response times or exhaust server resources. Second, artifact discovery operates over a dataset that grows monotonically as researchers deposit new projects. Database queries for public search must therefore remain performant under increasing data volume without requiring schema changes. The system shall handle concurrent uploads and conversion operations without unbounded resource consumption, and database query performance for public artifact discovery shall remain acceptable as the archive grows. The deployment model shall support vertical scaling and, where required, horizontal scaling without changes to the application code.

Interoperability: The system shall support at least the REFI-QDA open exchange format as both an import and export target, as this is the primary interoperability standard in the qualitative data ecosystem (Evers et al., 2020). The internal artifact representation shall be format-agnostic to support the addition of further format adapters without changes to the persistence schema or the project lifecycle logic.

Reliability: Qualitative research data is often the product of extended fieldwork that cannot be repeated. A system that silently corrupts or loses artifacts under partial failure is therefore unacceptable in this domain (Whyte & Tedds, 2011). Stable share links shall remain accessible until explicitly reset by the owner. Batch operations shall report per-item outcomes rather than failing silently. Multi-step operations shall be ordered and validated such that failures surface as structured errors to the caller rather than leaving

the system in an undetected inconsistent partial state (Gray & Reuter, 1992). Each API operation shall carry a trace identifier that propagates through the service and repository layers, enabling correlation of log entries for a single request (Kleppmann, 2017).

Usability and Accessibility: Core functions, such as archive search, file format conversion, and download of public artifacts, shall be available without account registration. Requiring an account only where the user explicitly wants to persist or share their own artifacts reduces friction for one-time or exploratory use cases (Nielsen, Jakob, 1994). Error responses from the API shall carry human-readable messages that allow callers to present actionable feedback to the user.

Privacy and Compliance: The system shall record explicit Terms of Service acceptance as a precondition for all functions that interact with user data or produce stored artifacts. Credentials and session tokens shall be transmitted only over encrypted connections (OWASP Foundation, 2024). Personal profile data shall be modifiable and deletable by the owning user (GDPR, 2016).

Distribution and Deployability: QDArchive integrates four external services. These are utilized for authentication, relational persistence, file storage, and format conversion and each of these introduces a deployment-time dependency. To prevent these dependencies from coupling the application code to specific service providers, each is accessed through a defined interface at a single integration point. This means that switching providers, or running the application against local service emulators during development, requires only configuration changes rather than code modifications (Wiggins, 2017). All environment-specific parameters (service URLs, API keys, and site URL) shall be supplied through environment variables, with no secrets committed to the version control repository.

3.5 Quality Attribute Scenarios

Quality requirements become evaluable when expressed in terms of concrete stimuli, affected system elements, and measurable responses. Selected quality attributes are formulated as structured scenarios (Bass et al., 2021). The scenario format was chosen over informal prose descriptions because it makes implicit architectural assumptions explicit. Rather than stating that the system is "secure" or "reliable", each scenario identifies which component bears responsibility and under what conditions the required property must hold.

Not all quality attributes are realized uniformly across the system. Security concerns are concentrated at the API and database tiers, where access control decisions are enforced at two independent layers so that neither alone constitutes a single point of failure (Saltzer & Schroeder, 1975). Maintainability, by contrast, is addressed structurally through the adapter layer. Isolating format-specific logic in a dedicated abstraction means that extending the system with new QDA formats requires changes in exactly one place (Parnas, 1972). Reliability is the responsibility of the service layer, which orchestrates multi-step operations and must guarantee that partial failures do not leave the system in an undetected inconsistent state (Gray & Reuter, 1992). The scenarios below make these design decisions evaluable by grounding each quality concern in a representative interaction.

Maintainability Scenario

Stimulus: A new QDA format becomes available in the conversion library.
Source: Updated version of the `@QDArchive/qdconvert` dependency.
Environment: Normal operation and maintenance.
Artifact: Conversion boundary around QDConvert.
Response: The new format becomes immediately available for import and export without requiring application-level code changes.
Response Measure: Zero modifications to the Share, Find, access control, versioning, or persistence subsystems.

This requirement led to the introduction of a dedicated conversion boundary around QDConvert. All format-specific parsing and serialization logic is encapsulated within the external conversion library, while the rest of the system operates exclusively on a format-agnostic internal representation. As a result, the addition of new CAQDAS formats does not propagate into the project lifecycle, persistence, or access control logic.

Security Scenario: Share Link Enumeration

Stimulus: An unauthenticated user requests a share link URL with a randomly generated token.
Source: External actor attempting enumeration.
Environment: Public deployment.
Artifact: Share link resolution endpoint.
Response: Access is denied and no project data is returned.
Response Measure: The token space provides sufficient entropy that the probability of a correct guess within a practical number of attempts is negligible (OWASP Foundation, 2024).

Security Scenario: Privilege Isolation

Stimulus: A bug in an API route handler causes it to skip an ownership check.
Source: Application defect.
Environment: Normal operation.
Artifact: Database tier.
Response: The row-level security policy on the relevant table rejects the query; no unauthorized data is returned.
Response Measure: The request returns an authorization error and no records belonging to other users are exposed.

The two security scenarios rely on deliberately redundant enforcement mechanisms. Application-level ownership checks and database-level row-level security policies operate independently, so bypassing one layer does not bypass the other. Share link tokens address a different threat surface because links must remain usable without authentication. Therefore they depend entirely on token unpredictability rather than session state.

Access Control Consistency Scenario

Stimulus: A project owner sets a project from private to public.
Source: Authenticated project owner.
Environment: Normal operation.
Artifact: Project update endpoint and share link subsystem.
Response: The project becomes discoverable via search; any previously issued share link is invalidated.
Response Measure: Subsequent requests to the old share link URL return an error. The project appears in public search results.

This scenario exists because the transition from private to public introduces a consistency obligation across two subsystems. A share link issued for a private project carries the implicit assumption that access is intentionally restricted. Once the project is public, that assumption no longer holds and the link becomes semantically meaningless. Invalidating it automatically prevents a situation in which an owner believes they have revoked selective access when in fact an old link continues to circulate.

Reliability Scenario

Stimulus: A failure occurs partway through a multi-step operation such as project version creation.
Source: Infrastructure disruption or application error.
Environment: Normal operation.
Artifact: Service and persistence layers.
Response: The system surfaces an error to the caller. No inconsistent partial state persists undetected.
Response Measure: No orphaned records or corrupted version entries remain after a failed operation.

The following scenario addresses a form of state consistency that spans an identity transition rather than a single operation. It is not a security scenario in the conventional sense, but a continuity requirement. The system must guarantee that a user's work survives a change in their authentication identity without any explicit migration step.

Anonymous Session Upgrade Scenario

Stimulus: A guest user completes a file conversion and subsequently registers an account within the same session.
Source: Anonymous user.
Environment: Normal operation.
Artifact: Authentication service and project ownership records.
Response: The anonymous session is upgraded to a registered account. Previously created projects remain accessible and are owned by the new account.
Response Measure: All projects created during the anonymous session are retrievable under the registered user's identity without any data migration step.

The reliability and session upgrade scenarios both concern state consistency, though at different scopes. The reliability scenario addresses intra-operation atomicity (Gray & Reuter, 1992). Version creation involves multiple dependent writes, and a partial failure must not leave the database in a state that appears internally valid but is logically corrupt. The session upgrade scenario concerns inter-operation continuity. The system must preserve ownership across an identity transition that the user experiences as a single seamless action. In both cases the design goal is that the system's observable state remains trustworthy. Either the operation succeeded completely, or it failed cleanly.

3.6 Requirements Summary

Table 3.1 maps the functional requirements to the user-facing solutions defined in the product specification and indicates their implementation status in the current system.

ID	Requirement (abbreviated)	Context
Find		
F1	Public search interface,	Find
F2–F3	Full-text search, public-only visibility	Find
F4–F6	Filtering (tags, author, date) and sorting	Find
Convert		
F7–F8	Conversion interface (guest + auth), ToS	Convert
F9	Format detection and validation	Convert
F10–F11	Dynamic target formats and download	Convert
F12–F13	Project creation and session-based ownership	Convert
F14	Anonymous session upgrade with ownership preservation	Convert
F15	Project visibility/editing for authenticated users	Convert
F16	Missing file detection and optional completion	Convert
Share (Account & Archive)		
F17–F21	Registration, login, logout, recovery, profile management	Share
F22	Project creation with metadata and visibility	Share
F23	Versioning support	Share
F24–F25	File upload and metadata management	Share
F26–F27	Deletion and project download	Share
Share (Publication & Access)		
F28	Secure share link (private, world-readable)	Share
F29	Share link regeneration	Share
F30	Public visibility and search discoverability	Share
Administrative Functions		
F31–F32	Review queue and ingestion decisions	Admin
Background Services		
F33–F35	External discovery, monitoring, backups	Infrastructure

Table 3.1: Mapping of functional requirements to solution areas

The requirements established in this chapter serve as the evaluation baseline for Chapter 6. The quality attributes defined in Section 3.4 and the scenarios in Section 3.5 provide the structural criteria against which the architecture and implementation are assessed. This includes, for example, the dual-layer access control model required by the coexistence of private, shared, and public artifacts, as well as the isolation of format conversion behind

a dedicated adapter. Any discrepancies between requirements and the current implementation are explicitly addressed in the evaluation (see chapter 6).

4 Architecture of the QDArchive Backend

This chapter presents the architecture of the QDArchive backend. The design is shaped by three quality concerns established in Chapter 3: (i) maintainability, which requires that format-specific logic can be extended without affecting the artifact lifecycle or access control subsystems; (ii) security, which demands that access restrictions hold independently of application-layer correctness, and (iii) reliability, which requires that multi-step operations either complete consistently or fail in a way that leaves no undetected partial state. The architectural decisions described in this chapter – the choice of a modular monolith, the domain-driven decomposition into bounded contexts, and the dual-layer access control model – are motivated by these concerns and by the operational constraints of a small-team, self-hosted deployment.

The chapter introduces the architectural style and its justification. In short, the modular decomposition derived from domain analysis, the domain model, and the key mechanisms that implement access control and data management. Where the current implementation deviates from or partially realizes the intended design, this is noted explicitly. A systematic evaluation against the stated quality requirements is provided in Chapter 6.

4.1 QDArchive as a Modular Monolith

QDArchive is implemented as a modular monolith, a single deployable unit whose internal structure is divided into well-defined, domain-aligned modules with explicit boundaries (Richards & Ford, 2020). The system is deployed in a containerized, self-hosted environment using Docker Compose, and integrates Supabase, an open-source backend platform built around PostgreSQL, for persistence, authentication and object storage.

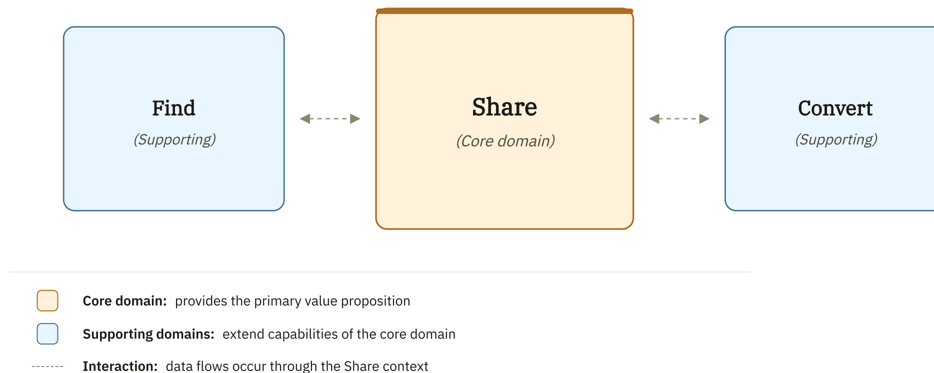

Figure 4.4: High-level modular structure of the QDArchive backend

The choice of a modular monolith over a microservice architecture is driven by three factors specific to the QDArchive context. First, the core operations of the system, including research project creation, versioning, access control enforcement, and publication state transitions, involve tightly coupled state across multiple domain entities. E.g., making a project public requires coordinated updates to the project record, its associated share links, and storage access policies. Distributing these operations across independent services would require either distributed transaction protocols such as two-phase commit, or an eventual consistency model in which intermediate states are temporarily visible across service boundaries. Both approaches introduce significant coordination overhead and fail-

ure surface without corresponding benefit at the current system scale (Newman, 2021). Also, QDArchive is operated by a small development team in a self-hosted environment. A microservice architecture would multiply the number of independently deployed and monitored components, on one hand increasing infrastructure complexity substantially whilst on the other complicating local development workflows (Newman, 2021). In addition, at the current stage of the project, the boundaries between functional areas are still being refined. Prematurely decomposing the system into independent services risks encoding incorrect boundaries that are costly to change later, whereas the modular monolith allows internal structure to evolve without requiring inter-service protocol changes (Fowler, 2015).

While a modular monolith avoids the operational complexity of distributed services, it still requires explicit mechanisms for maintaining architectural boundaries. Without intentional enforcement of internal structure, monolithic systems tend toward a state in which domain models are mixed together and assumptions leak across functional boundaries, making it impossible to maintain a coherent model of any single area (Vernon, 2016). The modular structure of QDArchive is designed to prevent this problem by treating module boundaries as explicit architectural elements.

At the same time, this structure is designed to preserve the option of future decomposition. Each module defines its responsibilities through internal service interfaces and communicates with other modules through controlled entry points rather then shared internal state. Individual modules could in principle be extracted into independent services if scalability or operational requirements change (Fowler, 2015; Newman, 2021). The Convert module is comparatively self-contained because it acts primarily as an integration boundary around the external QDConvert library. Format-specific parsing and serialization are delegated entirely to the library, while the module exposes a stable interface consumed by the Share module. The Find module is similarly isolated, as it operates primarily on a read-only projection of published archive data.

4.2 Domain-Driven Design

The architectural decomposition of QDArchive follows the principles of Domain-Driven Design (DDD), which emphasizes structuring software systems according to the underlying problem domain and its associated business concepts (Evans, 2004). Applied to QDArchive, this approach means that module boundaries, service responsibilities, and data structures are derived from the domain of qualitative research artifact management with technical concerns remaining secondary. This section describes the subdomain analysis that motivated the system's structure, the resulting bounded contexts and their relationships, the ubiquitous language, and the domain services defined within each context.

4.2.1 Subdomain Analysis

A key step in Domain-Driven Design is identifying and categorizing the subdomains that make up the system's business domain. A distinction is drawn between three types: (i) core subdomains, which provide competitive advantage and require complex, in-house implementation, (ii) generic subdomains, which are solved problems best addressed by existing solutions, and (iii) supporting subdomains, which are necessary but neither differentiating nor complex (Khononov, 2021).

Applied to QDArchive, this analysis yields the following classification. The Share subdomain, which encompasses artifact lifecycle management, versioning, access control, and

publication workflows, is the system's core subdomain. It represents the primary value proposition of QDArchive, which is managing QDA artifacts as structured, evolving research objects with explicit lifecycle states and multi-dimensional access control. This is what differentiates QDArchive from generic file storage systems, and it is where the highest engineering effort is concentrated. The Find and Convert subdomains are supporting subdomains. The Find subdomain provides search and discovery capabilities, but its logic is limited to read-only queries against a published view of the Share subdomain's data. It therefore supports the core domain while remaining relatively simple in its implementation (Khononov, 2021). The Convert subdomain handles the transformation of QDA artifacts between formats. Although format interoperability is a distinctive capability of QDArchive, and one directly motivated by the format lock-in problem identified in the related work, the Convert subdomain is classified as supporting subdomain for a deliberate reason. The actual conversion logic is entirely encapsulated in the external library QDConvert. The Convert subdomain provides an integration and orchestration layer that detects input formats, delegates transformation to the library, handles missing file references, and persists the result as a new project (representing a QDA artifact) in the Share context. Its responsibilities are centered on coordinating data transformation and persistence rather than on maintaining a complex domain model, and the engineering investment is concentrated in the adapter boundary instead of the conversion logic itself. Classifying Convert as a supporting subdomain therefore accurately reflects where the system's own complexity lies (Khononov, 2021).

In contrast, Identity and Storage are generic subdomains. Authentication and identity management are solved problems, addressed here by delegating entirely to the GoTrue authentication service provided by Supabase. Similarly, binary file storage is treated as a commodity infrastructure concern and is delegated to Supabase Storage. Generic subdomains are best addressed by adopting existing solutions instead of investing in inhouse implementations (Khononov, 2021). This classification directly justifies the thinness of the Identity and Storage modules, which contain no domain logic and serve purely as adapters to external infrastructure.

4.2.2 Bounded Contexts

While subdomains are discovered through domain analysis, bounded contexts are designed as part of the software solution (Khononov, 2021). In QDArchive, the decision was made to align bounded contexts one-to-one with the identified subdomains, resulting in the three primary bounded contexts Find, Convert, and Share. Identity and Storage are realized as supporting infrastructure modules rather than independent bounded contexts, as they contain no domain model of their own. This structure is illustrated in Figure 4.5.

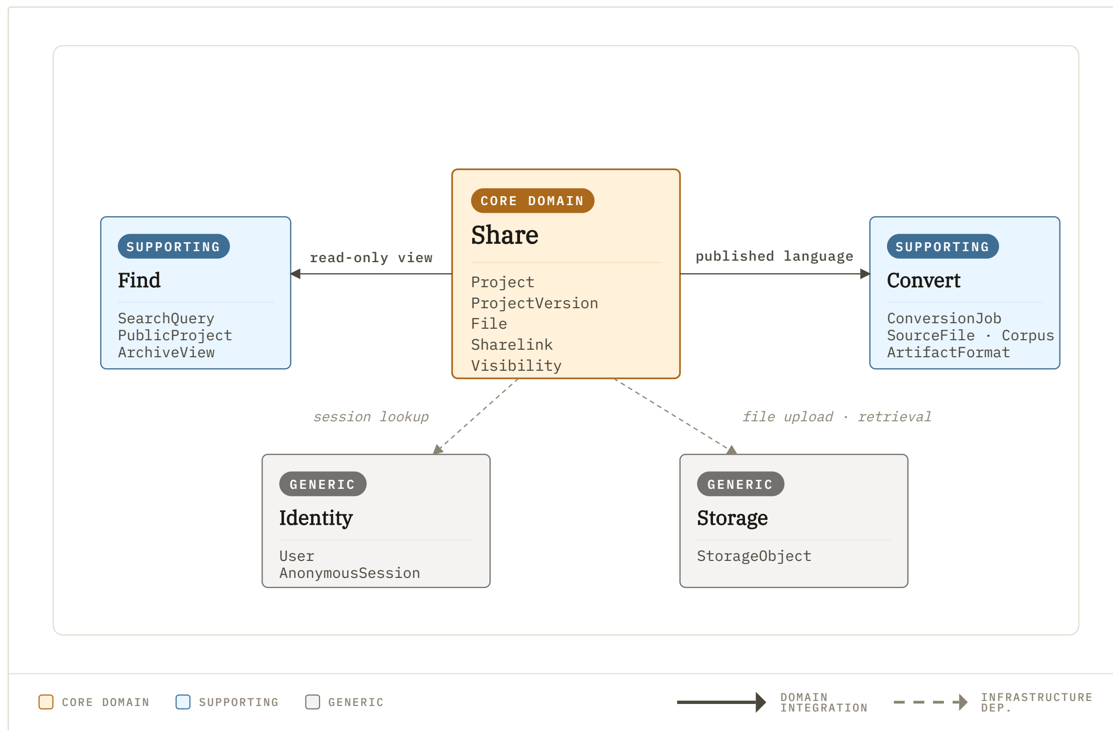

Figure 4.5: Bounded contexts and subdomain classification of QDArchive

The bounded contexts are organized around a central Share context, which owns the canonical representation of research projects and their lifecycle state. Both Find and Convert depend on interfaces exposed by Share but do not modify its internal model directly. This establishes a unidirectional dependency structure that prevents circular coupling between contexts and allows the central domain model to evolve without requiring coordinated changes across all modules.

4.2.3 Context Integration Patterns

Bounded contexts are not independent. They must integrate with one another to fulfill the system's overall function. Domain-Driven Design defines several patterns for characterizing these integrations (Khononov, 2021). In QDArchive, two integration relationships between bounded contexts are relevant.

The Convert context exposes a stable, format-agnostic corpus representation to the Share context. Imported QDA artifacts are transformed into this common internal model before they are handed over to Share for persistence and lifecycle management. By encapsulating all format-specific parsing and serialization logic, Convert shields the Share context from vendor-specific file structures and format differences. As a result, analytical content extracted from heterogeneous CAQDAS formats can be represented through a stable, format-agnostic archival model, while differences in vendor-specific file structures remain isolated within the Convert context and do not affect the domain logic of the Share context.

The Share context exposes a dedicated read-optimized interface for search consumers, the Archive View, which contains only publicly visible qualitative artifact data. This follows the Published Language pattern (Evans, 2004). The interface is designed around the needs of the Find context rather than the Share context's internal model, allowing the

Share module to evolve its data model without affecting search consumers as long as the published view contract is maintained.

4.2.4 Ubiquitous Language

A shared domain language is used consistently across the system to reduce ambiguity and align implementation with domain concepts (Evans, 2004; Khononov, 2021). The terminology has been refined throughout the development process and is reflected in module boundaries, API Design, and internal data structure.

Although the thesis discusses QDA artifacts from an archival perspective, the ubiquitous language of the system uses the term Project because this reflects both the terminology of common CAQDAS tools and the way researchers conceptualize their work. Within the system, a Project therefore represents the archival unit that is elsewhere described as a QDA artifact. Based on this terminology, the following key terms are used consistently throughout the system. A Project is the central research object managed within the system, consisting of structured analysis data, associated source files, and metadata. A Project Version is a specific snapshot of a Project capturing its state at a point in time. The Analysis Data File is the primary structured file of a Project, such as a QDPX or vendor-specific format, containing the analytical structure including codes, categories, and document references. Primary Data Files are raw qualitative data referenced by the Analysis Data File, such as interview transcripts, PDFs, media files, which may be embedded or external to the analysis data file. Generic Data Files are supplementary files not directly part of the analysis structure, such as documentation, licenses, or notes. A Conversion is the transformation of an analysis file between different QDA formats, which may result in partial data loss depending on format compatibility. Publication describes the act of making a Project accessible beyond its owner defining the visibility level and determining whether an artifact is discoverable. Visibility defines the accessibility state of a Project. Projects can be Private (owner-only), Shared (accessible via secure link), or Public (discoverable via search). A Share Link is a non-guessable link that provides controlled access to a Project or a specific Project version without making it publicly discoverable. Findability refers to the ability of a Project to appear in search results, which is restricted to publicly published Projects are included in the searchable index. Download Package is a user-defined export of an which may include only the Analysis Data File, the Analysis Data File with primary data, or all associated files bundled as an archive. Corpus Format is a internal system-specific, format-agnostic representation of a Project used to enable consistent storage and conversion across different QDA formats.

4.3 Domain Model and Persistence Design

The domain model of QDArchive captures the central concepts and consistency rules of the system. It is defined primarily within the Share module, which represents the core domain, and is structured around aggregates (Evans, 2004). These are clusters of entities and value objects that are modified together in order to maintain domain invariants. Because the system is centered around long-lived, versioned research projects, the domain structure also directly shapes the persistence strategy. This section therefore discusses both the aggregate design of the domain model and the representation of these aggregates within the relational persistence layer.

4.3.1 Aggregate Structure

The central aggregate of the Share domain models a versioned qualitative research artifact (Project) together with its associated files, publication state, and access restrictions. The aggregate root coordinates lifecycle transitions, version creation, and visibility changes, while subordinate entities represent historical Project states and their associated file references. Binary storage objects themselves are managed outside the aggregate boundary and are referenced only by identity.

The aggregate is designed according to four principles of aggregate design (Vernon, 2016). First, the aggregate protects the invariants within its boundary. This means that each version is associated with exactly one Project, receives a monotonically increasing version number upon creation, and cannot exist without a valid, non-deleted parent Project. Creating a new version produces a consistent representation of the Project state at a specific point in time. Analysis and Primary Data Files may be carried forward from the previous version or replaced, and Additional Data Files may be included optionally. Access permissions are always derived from the current visibility state of the artifact, ensuring that publication changes are immediately reflected in access control decisions. Second, the aggregate is kept small. Rather than loading a full Project with all versions and files in every operation, Project metadata, version lists, and file metadata are fetched independently, and the aggregate boundary is kept narrow around write operations. Third, other aggregates, specifically the User aggregate owned by the Identity module, are referenced by identity only (Evans, 2004). User information is therefore associated with the Project through identifiers rather than embedded directly within the aggregate, preserving clear ownership boundaries and reducing coupling between domains. Fourth, updates that span aggregate boundaries (e.g., the soft-delete cascade from Project to Project versions to files) are executed as ordered sequences of independent database operations rather than within a single atomic transaction, which is an accepted pattern for cross-aggregate coordination (Kleppmann, 2017; Vernon, 2016). The resulting consistency trade-offs and compensation strategy are discussed in Section 4.7.

4.3.2 Mapping to Persistence

The persistence model realizes the domain structure through a relational schema that separates Projects, Project versions, files, and sharing state into independently managed records. Projects are persisted as aggregate roots together with independently stored version records that capture the state of a Project at a specific point in time, including its version number, original format, and serialized analytical content.

Files are modeled separately from Project versions in order to support reuse across multiple versions. Associations between versions and files are represented explicitly through a linking structure that records the semantic role of each file within a version, distinguishing analysis files, converted outputs, primary research data, and supplementary material.

Sharing state is persisted separately from Project content, while publication state is represented directly as an attribute of the Project itself. Share Links are associated directly with their owning Project and structurally constrained such that a Project can have at most one active Share Link at a time.

All Project entities implement soft deletion, preserving auditability while excluding deleted records from normal application queries. Large binary objects are stored externally in ob-

ject storage, while the relational model maintains only the metadata and storage references required for retrieval.

4.3.3 Separation of Archival Structure and Analytical Content

A key persistence concern is how to store the analytical content of a QDA artifact. This structure includes the codes, coded segments, hierarchical code relationships, and document linkages that constitute the analytical work of any qualitative researcher. Therefore three broad options were considered.

A fully relational model would decompose this structure into normalized tables, e.g., a table for code, a table for coded segments, a memos table, and so on. This would make the analytical content queryable at the database level but would require the schema to model every relationship type defined by the QDA domain. The fundamental obstacle is format heterogeneity as different CAQDAS tools define different structural concepts, and the REFI-QDA standard itself does not enumerate every relationship variant. A relational schema would need to be revised each time the conversion library introduces support for a new format feature, coupling the persistence layer to the volatility of external format specifications. This directly contradicts the modifiability requirement that format-specific logic must not affect the archival or access control subsystems.

A graph database would match the graph-like structure of QDA analytical networks more naturally. This approach was rejected on operational grounds. A graph database would require an additional managed infrastructure component, is not provided by the Supabase stack, and would introduce a second query language and consistency model into a system that already relies on PostgreSQL for all other persistence concerns. The added complexity is not justified by the use cases of the current system, which does not require traversal of the analytical graph at query time.

A format-agnostic JSON snapshot, stored as a single serialized value per version, avoids both problems. The conversion library produces an internal corpus representation that is format-independent. This representation is stored whole and retrieved whole, without the application layer ever needing to inspect its internal structure for archival or access control purposes. The schema does not change when a new format is supported. The analytical content is preserved with full fidelity and can be re-exported to any supported format at any future point without re-parsing the original file. The accepted trade-off is that the analytical content is opaque to the database. Structured queries across codes, coded segments, or analytical relationships are outside the scope of what the persistence layer can support. This is an intentional scope constraint. The system's responsibility is preservation and re-export, not analytical querying. Search is provided at the Project-metadata level, which is sufficient for discovery and consistent with the capabilities of comparable qualitative data repositories (Karcher et al., 2016).

This decision reflects the separation of concerns between the two layers of the persistence model. The relational schema governs archival structure (ownership, versioning, access control, and file metadata), while the JSON snapshot governs analytical content (the Corpus produced by the conversion library). Neither layer needs to understand the other's internals.

4.4 Modularization

The internal structure of QDArchive is decomposed into modules that correspond directly to the bounded contexts defined in Section 4.2.2. Each module encapsulates a coherent set of domain responsibilities and exposes dedicated services through a defined interface, following the principle that modules should be independently understandable and evolvable without requiring knowledge of other modules' internals (Evans, 2004).

4.4.1 Module Structure

The system comprises of five modules named Find, Convert, Share, Identity, and Storage. The first three correspond to the primary bounded contexts of the domain. Identity and Storage are supporting infrastructure modules that provide shared capabilities consumed by the domain modules. The corresponding implementation structure and internal layering of these modules are described further in Chapter 5.

QDArchive follows a layered architecture consisting of an API layer, an application layer, and an infrastructure layer (Richards & Ford, 2020). The API layer handles incoming requests and translates them into application operations. Within the domain modules, application services orchestrate use cases by coordinating domain logic and infrastructure interactions. The infrastructure layer provides persistence through repository classes and delegates file operations to the Storage module. This separation reduces coupling between domain logic and technical infrastructure concerns. Application services remain independent of HTTP handling, while repository classes encapsulate persistence-specific behavior.

4.4.2 Module Responsibilities

The five modules form a deliberate dependency structure rather than a flat collection of independent components. The Share module represents the core domain and owns project lifecycle management, versioning, publication state, and access control. The Convert and Find modules provide supporting capabilities. Convert encapsulates interoperability concerns through integration with QDConvert, while Find supports discovery of publicly available Projects through a dedicated read model. Identity and Storage act as infrastructure adapters shared across the domain modules without containing domain logic themselves. This separation ensures that changes in one functional area have minimal impact on others. Changes to external QDA formats primarily affect Convert, evolving discovery requirements affect Find, and infrastructure substitutions remain confined to Identity and Storage, while the core lifecycle and authorization logic of Share remains stable.

4.5 Application and Service Model

QDArchive organizes application behavior around dedicated application services that coordinate domain workflows across modules and infrastructure components (Fowler, 2002). These services implement the operational use cases of the system, including Project lifecycle management, versioning workflows, access control decisions, and format conversion processes.

Rather than embedding workflow logic directly into API endpoints or persistence components, the system centralizes orchestration within the application layer and delegates

persistence and infrastructure concerns to dedicated adapters. The following sections describe the resulting separation of responsibilities, the coordination model between services, and the approach used for dependency composition.

4.5.1 Layered Responsibility Separation

QDArchive applies a layered application structure that separates request handling, orchestration, and infrastructure access into distinct responsibilities (Richards & Ford, 2020). This organization ensures that business rules remain independent from transport protocols and persistence-specific implementation details, whilst infrastructure concerns remain isolated from the domain model.

The outer layer consists of the API layer, which acts as the boundary between external clients and the application. Its responsibilities are limited to request handling, authentication, input validation, and translation between HTTP representations and application-level operations. Application workflows are not implemented at this layer. Instead, requests are delegated to application services responsible for coordinating the required use cases.

The application layer forms the operational core of the system. Application services orchestrate workflows that span multiple entities, repositories, and infrastructure adapters. This includes coordinating Project lifecycle operations, version creation, file management, access-control decisions, and conversion workflows. Complex operations are therefore implemented through collaboration between dedicated services instead of being concentrated in a single component. This keeps workflow logic centralized and prevents business rules from being duplicated across API endpoints or persistence components.

The infrastructure layer provides persistence and integration capabilities required by the application layer. Repository components encapsulate database access, while dedicated infrastructure adapters manage interactions with external systems such as object storage, authentication providers, and the conversion library. These components contain no application coordination logic themselves and are used exclusively through the application layer.

Dependencies follow a strictly inward-facing direction from the API layer to the application layer and finally to infrastructure components. Infrastructure concerns therefore do not propagate into domain coordination logic, and repositories remain focused on data access rather than workflow orchestration. This separation improves maintainability and testability by allowing application services to be reasoned about independently of HTTP handling or persistence-specific implementation details.

4.5.2 Dependency Composition

QDArchive uses explicit dependency composition instead of a centralized dependency injection container (Fowler, 2002). Application services, repositories, and infrastructure adapters are assembled at clearly defined composition points, making dependencies visible and avoiding hidden runtime wiring.

This approach keeps the runtime model comparatively simple and aligns with the modular monolith architecture of the system. Dependency relationships remain localized and explicit, which improves understandability and reduces the risk of unintended coupling between modules.

The composition model is also influenced by the system's security architecture. A new Supabase client is created for each request using the session information contained in the request cookies. Persistence-related dependencies are then composed around this client. This ensures that row-level security policies are evaluated against the identity of the currently authenticated user.

Explicit dependency composition also improves testability. By depending on abstractions and receiving infrastructure dependencies through composition, application services can be tested in isolation using test doubles for persistence and storage concerns

4.6 Supporting Architectural Mechanisms

Beyond the modular structure and service model, several architectural mechanisms address concerns that span multiple modules. These mechanisms implement requirements related to data management, access control, as well as the processing model, and shape key structural decisions throughout the system.

4.6.1 Data Management

QDArchive separates structured metadata from binary project content across two storage backends. This separation reflects the dual nature of qualitative research projects as both structured research objects and potentially large binary files. Relational storage provides the consistency guarantees and query capabilities required for metadata and access control, while object storage handles binary content without introducing blob overhead into the relational model. The concrete realization of this separation is described in Section 5.1.

4.6.2 Access Control Model

Access control in QDArchive enforced across two complementary layers. At the database level through PostgreSQL's row-level security policies, and at the application level through ownership checks in service methods and authentication guards at the API boundary. Permissions are evaluated using artifact-specific lifecycle information such as ownership, publication status, and sharing configuration, alongside role-based considerations. The concrete implementation of both layers is described in Section 5.4.

4.6.3 Processing Model

In the current implementation, all operations including format conversion are executed synchronously within the HTTP request lifecycle. This simplifies the implementation and avoids the operational overhead of a job queue, but means that long-running conversion operations block the response for the duration of processing. For the file sizes typical in the current system this is acceptable, but it represents a scalability constraint as artifact size or concurrent usage grows. An asynchronous execution model (in which a conversion job is enqueued and the client polls for or is notified of the result) would improve responsiveness and decouple processing time from request latency. This is identified as a planned improvement and is discussed further in Chapter 6.

4.7 Architectural Trade-offs

Every architectural decision involves trade-offs between competing quality attributes and operational constraints. The trade-offs discussed in this section reflect a consistent architectural priority throughout QDArchive. Preserving domain consistency and operational

simplicity therefore takes precedence over maximizing infrastructure flexibility or scalability at the current stage of the system.

This prioritization follows directly from the system context established throughout the previous chapters. QDArchive is designed as a research-oriented archival platform operated in a small-team, self-hosted environment, where correctness of publication and versioning workflows, access control, and artifact preservation is more critical than optimizing for large-scale distributed operation. Architectural decisions such as the use of a modular monolith, synchronous processing, and infrastructure delegation to Supabase therefore intentionally favor reduced operational complexity and stronger consistency boundaries over early optimization for horizontal scalability.

Limitations arising from incomplete realization of the intended design are addressed separately in Chapter 6.

4.7.1 Infrastructure Dependency on Supabase

QDArchive delegates authentication, database access, object storage, and API gateway functionality to a self-hosted Supabase stack instead of assembling these components independently. This reduces operational complexity significantly. A single Docker Compose configuration manages eleven infrastructure containers including PostgreSQL, GoTrue for authentication, PostgREST for the database API, Kong as the API gateway, and MinIO-backed object storage. The alternative, consisting of independently configuring and maintaining each of these components, would require substantially more infrastructure expertise and ongoing maintenance effort for any small development team.

The trade-off is reduced flexibility and increased coupling to a specific infrastructure stack. The Kong version bundled with Supabase cannot be upgraded independently, and certain PostgreSQL extensions or configurations may conflict with the assumptions made by Supabase's internal components. Migrating away from Supabase at a later point would require replacing multiple infrastructure layers simultaneously. This dependency was accepted as a deliberate compromise, prioritizing operational simplicity and development velocity over long-term infrastructure independence. The risk of lock-in is partially mitigated by the architectural separation described in Section 4.4. The Identity and Storage modules each encapsulate all Supabase-specific API calls behind a defined interface, so that the domain logic in the Share and Convert modules is not directly coupled to Supabase constructs. Replacing the authentication or storage backend would require changes confined to these two modules, limiting the scope of change across the remainder of the system. The database dependency is less well-insulated, since RLS policies are PostgreSQL-specific and would need to be reimplemented against a different access control mechanism if the database were replaced.

4.7.2 Database-level versus Application-level Access Control

Enforcing access control through PostgreSQL's row-level security provides strong guarantees beyond those achievable through application-layer checks alone. Policies are evaluated by the database engine and cannot be bypassed by application-layer bugs or missing authorization checks. This is particularly valuable in a system where multiple code paths may access the same data under different authorization contexts, this includes in particular standard domain operations, administrative operations, and guest session handling. The trade-off is that RLS policies are defined in SQL migrations and evaluated at the

database level, which makes them harder to test in isolation than application-layer authorization logic. Testing RLS policies requires a running database instance with the correct schema and user sessions, instead of simple unit tests. Additionally, the logic for access decisions is now split across two layers (database policies and application-layer checks) which requires careful coordination to ensure consistency. In QDArchive, this split is managed through a clear convention. Row-level security enforces ownership and visibility constraints at the database level, while the application layer performs additional ownership validation and handles higher-level business rules such as Share Link validation and permission management.

4.7.3 Synchronous Request Processing

As described in Section 4.6, all operations including format conversion are currently executed synchronously within the HTTP request lifecycle. The trade-off accepted here is implementation simplicity and debuggability at the cost of responsiveness under load. There is no job queue, no worker process, and no need for the client to poll for results. For the file sizes and usage volumes of the current system this seems acceptable. The concrete path to an asynchronous model would involve introducing a job queue, e.g. via a dedicated worker process or a database-backed queue table, and replacing the synchronous conversion call with an enqueue operation. The client would subsequently poll a status endpoint or receive a webhook notification on completion. This transition is architecturally straightforward given the existing module boundaries, since the Convert module already encapsulates the conversion operation behind a single service call that could be moved off the request thread without changes to the surrounding modules.

4.7.4 Dependency Direction between Share and Convert

Although format interoperability is a distinctive capability of QDArchive, the Convert subdomain is classified as supporting rather than core. Conversion itself is not the primary business value of the system, but a supporting capability that enables archival and sharing workflows.

This results in a limited dependency from the Share module to the Convert module, since archival operations rely on format parsing and serialization. The coupling remains intentionally narrow. Convert encapsulates all format-specific logic behind a stable service interface and does not depend on archival lifecycle management or domain state itself.

4.7.5 Compensating Transactions and Saga Coordination

The Supabase JavaScript client does not expose database-level transactions to application code. Multi-step write operations, e.g. Project creation, version creation, and the soft-delete cascade, are therefore implemented as ordered sequences of independent database calls rather than as atomic units (Supabase, 2021). This is an accepted pattern for cross-aggregate coordination in Domain-Driven Design (Vernon, 2016), but it introduces a concrete risk. A failure partway through a sequence leaves the system in a partially applied state. For the two most critical sequences (Project creation and Projectversion creation) this risk is mitigated by a Saga-based compensation mechanism (Garcia-Molina & Salem, 1987). Each step registers a compensating action that is executed in reverse if a subsequent step fails, restoring database records and any already-uploaded storage objects to a consistent state. This approach comes with two consequences. First, compensation logic must be written and maintained alongside the forward logic, increasing implementation

complexity for every covered operation. Second, not all multi-step sequences in the current implementation are covered by this mechanism, but only the two most critical paths. Operations outside this scope remain vulnerable to partial failure, which is an accepted limitation given the append-oriented data model. Incomplete writes result in isolated intermediate states while preserving the structural integrity of the underlying data. These states can be identified and cleaned up without affecting other records. The implications of the uncovered cases are discussed in Chapter 6.

The architectural decisions described in this chapter establish the structural foundation for the implementation. Chapter 5 describes how these decisions are realized in a working system, with particular attention to three implementation choices that most directly reflect the architectural intent. The separation between archival structure and analytical content in the persistence layer, the layered service decomposition that keeps domain logic testable and infrastructure-independent, and the dual-layer access control mechanism that operationalizes the lifecycle-aware authorization model.

5 Implementation

This chapter describes how the architectural design introduced in Chapter 4 was realized in a working system. The central implementation challenge was not the replication of standard web backend patterns, but their careful adaptation to the specific demands of qualitative research data. Projects that combine structured analytical content with large binary files, access control that must evolve with the publication and versioning workflows of each Project, and a format landscape dominated by proprietary, incompatible representations.

Three decisions shaped the implementation throughout: (i) the separation between archival structure and analytical content, in which the graph-like structure of Projects representing QDA artifacts is preserved as a format-agnostic serialized snapshot rather than decomposed into relational tables, (ii) strict layering of API, service, and repository classes, and (iii) a dual-layer access control model in which database- and application-level checks are deliberately redundant. The goal of this chapter is to demonstrate where these decisions are visible in the implementation and how they support the quality attributes of modifiability, security, and reliability established in Chapter 3.

5.1 Technology Stack

The implementation follows a modular monolithic architecture, as described in Chapter 4 using a TypeScript-based Node.js runtime environment, with Supabase serving as the integrated backend platform that provides database, authentication, and storage services.

TypeScript was chosen as the primary language for implementing the backend to support explicit module boundaries and type-safe service interfaces to define clear contracts between components. Service classes expose typed method signatures, repository layers specify the structure of the data they operate on, and domain transfer objects define consistent data shapes across module boundaries. This becomes particularly useful when working with heterogeneous QDA artifacts, where differences in structure can easily lead to subtle errors. By enforcing these contracts at compile time, many inconsistencies can be detected early before they manifest during execution. In addition, the compiler is configured in strict mode, requiring incomplete or nullable data to be handled explicitly. This is an important property in an archival context where uploaded datasets may contain missing or partially defined information (Microsoft, 2024)

The runtime environment is Node.js, integrated via Next.js 16. As the system is primarily I/O-bound and performs relatively little CPU-intensive computation, Node.js's single-threaded, event-loop-based model is well suited to the workload. It efficiently handles concurrent operations such as file uploads, database queries, and storage interactions without the overhead associated with thread-based concurrency (Node.js, 2026a). The ecosystem surrounding Node.js also provides mature libraries for archive generation (`archiver`), UUID generation, and multipart form handling, all of which are used in the implementation. Next.js serves as the application framework, providing a unified environment for both frontend and backend execution (Vercel, 2026). Its server-side route handler API allows backend logic to run within the same Node.js process as the frontend, eliminating the need for a separately deployed backend service. This aligns naturally with the modular monolith approach, enabling deployment as a single unit while preserving clear internal module boundaries. Compared to a split frontend–backend architecture, this reduces operational complexity and simplifies deployment without sacrificing separa-

tion of concerns, which is instead enforced at the code level through module structure and TypeScript interfaces.

Supabase serves as the core backend infrastructure and provides a tightly integrated set of services(Supabase, n.d.):

- PostgreSQL (v15) as the primary relational database
- GoTrue for authentication and JWT-based session management
- Supabase Storage (S3-compatible) for binary artifact storage
- Kong API Gateway for internal request routing within the Supabase stack
- PostgREST for database access via the Supabase JavaScript client

PostgreSQL was selected as the database system as it is provided natively by Supabase and aligns well with the consistency requirements of the domain model. Its ACID guarantees and declarative integrity constraints support reliable management of the relationships between Projects, versions, and files (PostgreSQL, 2026d, 2026g). In addition, PostgreSQL's row-level security mechanism allows access control policies to be enforced directly at the database layer, independently of application logic (PostgreSQL, 2026f). The choice of a relational model over alternatives such as a graph database is discussed in Section 4.3.3.

The system separates relational metadata from binary artifact content. Relational data such as Project metadata, versioning information, and access control rules are stored in PostgreSQL, while binary files are managed through Supabase Storage. Semi-structured analytical content is stored using PostgreSQL JSON types. The `artifact_data` column contains serialized corpus snapshots produced by the conversion library, while `source_metadata` stores provenance information. This snapshot-based approach preserves analytical content in a format-agnostic representation without coupling the persistence schema to specific CAQDAS formats as decribed in 4.3.3.

Authentication is handled by GoTrue, which issues JWTs that are validated on each request. Access control is enforced both at the application level and directly within the database via RLS policies, which evaluate the JWT claims to determine row-level permissions.

Using Supabase as a managed infrastructure layer reduces operational complexity significantly. Database provisioning, connection pooling, storage bucket management, and authentication flows are provided out of the box and configured through migrations and environment variables rather than custom infrastructure code. The trade-off is reduced flexibility in areas where Supabase imposes constraints on configuration. E.g., the Kong gateway configuration is constrained by the version bundled with the Supabase self-hosted stack, and advanced database extensions must be explicitly enabled through migration scripts. In the current early stage of the Project, the productivity gains of the managed approach outweigh these limitations.

5.2 System Structure

This section describes how the architectural decomposition outlined in Chapter 4 is realized in the implementation. It first maps bounded contexts to concrete modules and directory structures, and then explains the layered organization used within each module.

5.2.1 Module Mapping

The repository structure gives each bounded context from Section 4.2 a physical location in the codebase. Each architectural module is mapped to a dedicated directory under `modules/`, reflecting the bounded-context structure introduced in Chapter 4. Mapping bounded contexts directly to top-level directories was chosen to make architectural boundaries visible in everyday development workflows. This structure was chosen over a purely technical decomposition (e.g. controllers, services, repositories as global top-level folders) to keep domain concepts cohesive and reduce coupling between unrelated parts of the system. Without physical separation, the modular monolith would risk degrading into a logically coupled codebase despite its conceptual decomposition. While these boundaries are enforced primarily by convention, the directory structure makes cross-module dependencies explicit. Table 5.2 shows the correspondence between domain concepts and their implementation.

Subdomain	Bounded Context	Module	Directory
Share (core)	Share	Share	`modules/share/`
Find (supporting)	Find	Find	`modules/search/`
Convert (supporting)	Convert	Convert	`modules/convert/`
Identity (generic)	—	Identity	`modules/identity/`
Storage (generic)	—	Storage	`modules/storage/`

Table 5.2: Mapping of domain concepts to implementation modules

The simplified top-level directory structure of the backend-relevant parts of the application is shown below:

```
web/
|-- app/
|   \-- (protected)/
|       \-- api/
|           |-- archive/
|           |-- auth/
|           |-- convert/
|           |-- projects/
|           |-- shareLinks/
|           \-- user/
|-- lib/
|-- modules/
|   |-- share/
|   |   |-- file/
|   |   |-- project/
|   |   |-- projectVersion/
|   |   \-- shareLink/
|   |-- convert/
|   |-- identity/
|   |-- find/
|   \-- storage/
|-- models/
\-- utils/
```

The repository structure also reflects several implementation-specific decisions that are not visible at the architectural level alone.

- The Share module is internally subdivided into the `Project`, `ProjectVersion`, and `File` submodules, reflecting the differing responsibilities of Project metadata, version snapshots, and binary file handling.
- The Find module accesses only the `archive_projects` database view and does not participate in write-oriented lifecycle management.
- The Convert module acts primarily as an adapter around the external QDConvert library and isolates format-specific parsing concerns from the remainder of the system.
- Identity and Storage encapsulate Supabase-specific integration logic and thereby reduce direct infrastructure coupling within the domain modules.

In addition to the domain modules, the codebase includes several shared supporting directories. The `models/` directory defines shared TypeScript interfaces, enums, and data transfer objects for all domain objects (e.g., `Project`, `ProjectVersion`, `ProjectVersionFile`). These types carry minimal infrastructure dependencies and serve as the shared vocabulary imported across module and layer boundaries. Centralizing these shared types avoids duplicate representations of core domain concepts across modules while still keeping infrastructure dependencies minimal.

5.2.2 Layered Separation

Within each module, requests pass through a consistent layered structure from route handlers to services and persistence components. The layered structure deliberately concentrates domain coordination in the service layer rather than distributing business rules across route handlers or repositories. This keeps infrastructure concerns isolated from lifecycle logic and allows ownership and consistency rules to remain testable independently of HTTP and persistence details. Figure 5.6 illustrates the request flow.

A layered architecture was chosen over directly coupling route handlers to persistence operations in order to isolate HTTP concerns, domain coordination, and infrastructure access from each other. This reduces the impact of infrastructure changes on application logic and keeps business rules centrally testable.

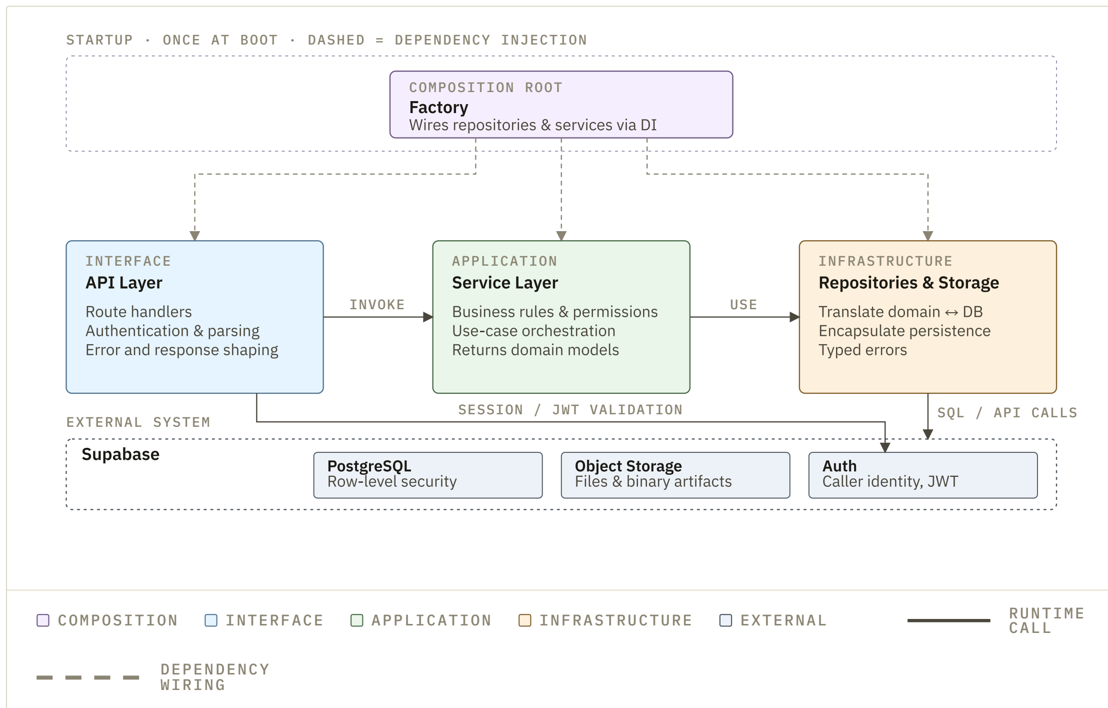

Figure 5.6: Layered request flow and dependency composition throughout the QDArchive backend

The outermost layer consists of Next.js route handlers located under `app/(protected)/api/`. These handlers are responsible for request parsing, authentication, service instantiation, and HTTP response generation. Supabase-backed helper functions perform Authentication before requests are delegated to the service layer. Domain logic itself is not implemented in route handlers. This prevents business rules from being duplicated across endpoints and keeps application behavior independent from HTTP-specific concerns. The `withErrorHandling` wrapper centralizes Error handling and assigns trace identifiers.

Application logic is implemented in service classes (`**.service.ts`), which coordinate repositories and infrastructure adapters while enforcing business rules and ownership constraints. Services also orchestrate multi-step workflows such as Project and version creation. External dependencies, including repositories, the `StorageService`, and the `ConvertService`, are passed through constructor injection, keeping the service layer independent from direct persistence concerns.

Object storage is separated from relational repositories because binary file management follows fundamentally different access and lifecycle semantics than relational metadata persistence. Persistence and storage access are encapsulated in repository classes and the `StorageService`. Repositories translate domain-level operations into Supabase query-builder calls and return typed domain objects, while the `StorageService` encapsulates object-storage operations such as file upload, retrieval, and signed URL generation. The repository abstraction also decouples the service layer from the specific query API of Supabase, limiting persistence-specific changes to a small number of infrastructure classes. Neither layer contains domain logic. Both act as infrastructure adapters used by the service layer.

A full dependency injection framework was intentionally avoided to keep the runtime model simple and explicit for a comparatively small modular monolith. Because the codebase does not use a dependency injection container, factory functions act as lightweight composition roots (Fowler, 2004). E.g., the `ProjectService` factory instantiates the repositories and infrastructure adapters required for a project and wires them together with request-scoped Supabase clients. Therefore, service construction remains transparent because dependencies are instantiated and wired together in a single location. The request-specific Supabase clients carry the authenticated user's session information, allowing row-level security policies to be evaluated against the correct user identity.

This layered structure supports both maintainability and testability. The domain logic in service classes can be unit-tested by substituting repository and storage dependencies with test doubles, without requiring a live Supabase instance. At the same time, the clear layer boundary prevents infrastructure concerns, such as query construction, storage path conventions, signed URL expiry, from leaking into business logic, and prevents domain logic from being fragmented across route handlers.

5.3 Data Model and Persistence

The persistence layer is implemented as a relational schema in PostgreSQL. The schema models Projects, versions, files, users, and access-control entities together with their relationships, while separating structured metadata from binary artifact storage. Analytical content is preserved as serialized JSON snapshots within the relational model, whereas binary files are stored externally in S3-compatible object storage and referenced through metadata records. Separating analytical structure from binary source material reflects the dual nature of CAQDAS artifacts. The analytical model must remain format-agnostic and versionable, while binary sources require scalable object storage semantics and may be only partially available in imported corpora.

Figure 5.7 illustrates the resulting relationships between the core entities of the persistence model.

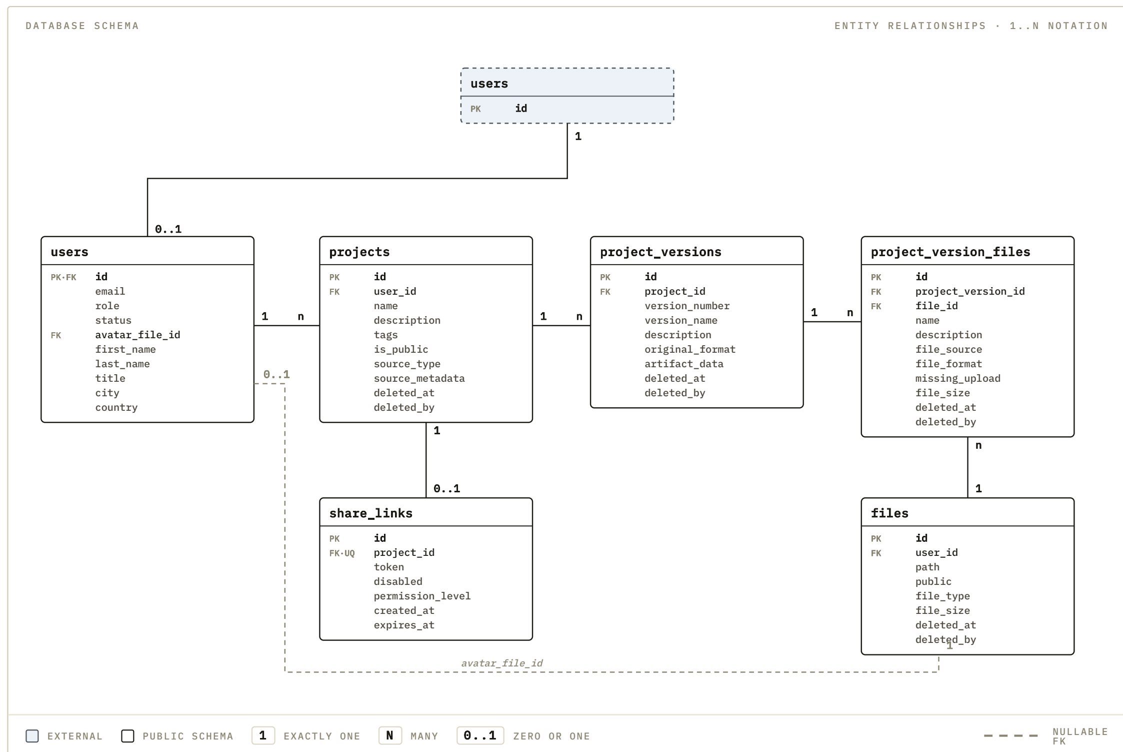

Figure 5.7: Entity-Relationship Diagram of the core database schema

5.3.1 Core Tables

The schema models the core domain entities and their relationships, with particular emphasis on the versioning of qualitative data artifacts. The tables are described in order of the aggregate hierarchy.

The `users` table extends the authentication model provided by GoTrue. Each row is linked to the corresponding entry in the internal `auth.users` table via a foreign key on the primary key, with cascading deletion. The table stores the application-level role of the user (`user`, `admin`, or `reviewer`), an account status (`created`, `confirmed`, `active`, `suspended`, or `removed`), and extended profile fields added in a later migration: `first_name`, `last_name`, `title`, `city`, `country`, `avatar_url`, and a foreign key reference to an optional avatar file in the `files` table.

The `projects` table represents the top-level artifact in the archive. Each Project is owned either by a registered user or an anonymous guest session (both identified via `user_id`) which enables unauthenticated conversion workflows without requiring registration. A Project carries a visibility flag (`is_public`), a `tags` array, and `source_type` discriminator (`user_upload`, `qdc_import`, or `web_import`) that records how the artifact entered the system. The `source_metadata` column stores additional provenance information, such as import origin or conversion parameters, in `JSONB` format. Unlike `artifact_data`, which is always read and written as a whole, individual provenance fields may need to be queried or filtered independently, making `JSONB`'s indexing capabilities the appropriate choice over a fixed schema. A generated `tsvector` column (`search_vector`) is maintained automatically from the Project name and description and used for full-text search, described further in Section 5.3.2.

The project_versions table is the central entity of the versioning model and directly reflects the iterative nature of qualitative data analysis. Each row represents one discrete state of a Project at a point in time, linked to its parent via project_id and carry a monotonically increasing version_number, a human-readable version_name, and a textual description. The original_format column records the QDA file format (e.g. qdpx or cprs) in which the artifact was originally submitted. The artifact_data column stores the serialized corpus representation produced by QDConvert as a plain JSON value rather than JSONB (PostgreSQL, 2026c). Because this column is always read and written as a whole snapshot, indexing individual fields would introduce write overhead without query benefit. Plain JSON is therefore the appropriate choice. Why relational decomposition and a graph database were both rejected in favor of an opaque serialized representation is discussed in Section 4.3.3. The versioning model reflects a concrete QDA workflow. An initial upload creates the first version, and subsequent changes, such as revised categories, merged codebooks, added coded segments, are captured as new versions. Each version may carry over files from the previous one, copy a subset, or introduce an entirely new analysis file. This explicit history provides the traceability characteristic of rigorous qualitative research practice (Saldaña, 2021).

The files table stores metadata about objects in object storage that are treated as opaque content Each row records the storage path of the corresponding object in the S3-compatible bucket, the owning user, a visibility flag, and a file_type discriminator (project_related, avatar, or user_uploaded). The table does not store file names directly. Naming is delegated to the junction table. Files marked as project_related are those attached to Project versions. The avatar entries are linked from the users table.

The project_version_files table maps files to specific Project versions and extends the pure many-to-many relationship with domain-relevant attributes. A name and description for display purposes, a file_format string, a file_size in bytes, and a missing_upload flag that indicates whether a file referenced in the corpus metadata was not included in the uploaded archive and is therefore unavailable for download. The central column is file_source, an enumerated type that classifies each file's role within the version:

- analysis_data_file: the original QDA Project file as uploaded
- converted_analysis_data_file: the file produced by the conversion pipeline in the target format
- primary_data_file: raw qualitative sources (e.g. interview transcripts, audio or video recordings) referenced inside the corpus
- generic_data_file: any additional attachment uploaded manually by the user

This classification allows the download and conversion logic to identify the relevant file for each operation without inspecting the binary content. It also makes the distinction between researcher-authored analysis and raw source material explicit at the data model level.

The Share_links table supports controlled external access to private Projects. Each Project may have at most one Share Link, enforced through the primary key on project_id. A Share Link stores a UUID-based access token, a disabled flag, and a permission_level (view, review, or edit).

5.3.2 Indexing and Search

The relational schema not only stores archival data but also supports public discovery and retrieval operations. To enable efficient search over Projects and metadata, the implementation uses several targeted indexing strategies. The `Projects` table includes a generated column `search_vector` of type `tsvector` defined as:

```
generated always as (
  to_tsvector('english',
    coalesce(name, '') || ' ' || coalesce(description, ''))
) stored
```

The column is computed automatically by PostgreSQL on insert and update, combining the Project name and description into a searchable representation. The `english` configuration applies stemming and stop-word removal, so that queries for "coding" also match related forms such as "coded" or "codes". A GIN (Generalized Inverted Index) index is defined on `search_vector` to support efficient full-text queries using the `@@` operator (PostgreSQL, 2026a). GIN is particularly suited to this use case, as each document maps to multiple lexemes, making multi-valued indexing more effective than traditional B-tree structures.

There are several supplementary indexes alongside the search index. Partial B-tree indexes on `created_at` and `updated_at` filtered to `is_public = true` support efficient time-ordered queries over the public archive without scanning private rows. A GIN index on the `tags` array column enables tag-based filtering without a separate tag table(PostgreSQL, 2026b). B-tree indexes on `deleted_at` in the `files` and `Projects` tables support fast filtering of soft-deleted records in cleanup queries. These indexes were added incrementally as query patterns became apparent during development and represent a pragmatic rather than exhaustive optimization.

5.3.3 Integrity Enforcement

Data integrity is enforced through a combination of declarative database constraints, soft-deletion rules, and procedural mechanisms such as triggers. The Implementation separates concerns between database and application logic. The database enforces schema-level invariants directly, while context-dependent validation is handled in the application layer.

Foreign key constraints enforce the relationships between Projects, Projectversions, Files, and Users shown in Figure 5.7. They ensure referential integrity by preventing orphaned records. The `users` table is synchronized with `auth.users` through cascading deletion, to ensure that removing an authentication identity also removes the corresponding application-level user record.

For domain entities such as Projects, versions, and files, physical cascading deletion is intentionally not used. Instead, deletions are handled through application-level soft deletion, which sets `deleted_at` instead of removing records from the database. This approach was chosen to support traceability in qualitative research archiving. A version removed from a researcher's active view may still be relevant for understanding the history of an analysis. Soft-deletion also simplifies error recovery, since no data is permanently lost until an explicit cleanup operation is run. Share Links are treated differently because they function only as transient access tokens. They are explicitly removed at the application layer in two scenarios: when a Project is soft-deleted, and when a Project is made publicly visible.

A PostgreSQL trigger ensures synchronization between the authentication system and the application schema. The trigger fires after each `INSERT` on `auth.users` and creates a corresponding row in `public.users` with the new user's UUID and email address. This keeps the application-level user record synchronized with the authentication system without requiring a separate write after registration.

An exception applies to anonymous sign-ins. Because anonymous sessions carry no email address and represent a transient identity, the trigger explicitly skips row creation for these users. Anonymous users therefore have no corresponding entry in `public.users` and are identified solely through their `auth.uid()`.

Because multi-step write operations are not executed within a single database transaction (see Section 5.1), these integrity mechanisms play a critical role in preventing invalid intermediate states. While they do not guarantee atomicity across multiple operations, they ensure that any persisted state remains structurally valid and recoverable.

5.4 Access Control

Access control in QDArchive is implemented at two complementary layers. The database layer uses PostgreSQL's row-level security (RLS) to enforce access invariants close to the data, while the application layer applies additional ownership checks in service methods and guards API routes through authentication helpers. This dual-layer design ensures that even in the presence of application bugs, the database will not return data to an unauthorized caller.

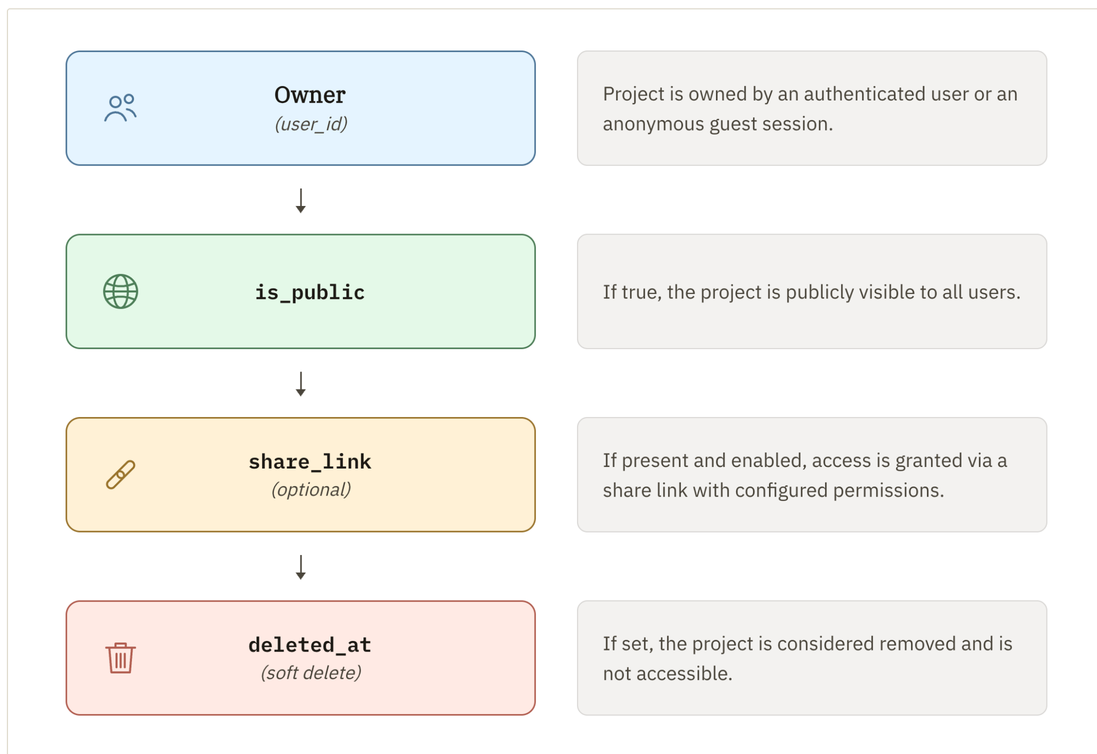

Figure 5.8: Factors determining effective Project visibility and access in QDArchive

Access control in QDArchive is determined by a combination of ownership, visibility state, Share Link configuration, and soft deletion. Public visibility (`is_public`) allows unrestricted access to Project metadata and associated files. For private Projects, access is limited to the owner or users with a valid Share Link. Share Links provide controlled external access without making a Project publicly discoverable. If a Project is soft-deleted (`deleted_at` set), it is excluded from all access paths regardless of other conditions

5.4.1 Role-Based Access

The system defines three user roles as a PostgreSQL enum: `user`, `admin`, and `reviewer`. These roles are stored in the `public.users` table alongside a five-state status enum (`created`, `confirmed`, `active`, `suspended`, `removed`) and are propagated into JSON Web Tokens issued by the GoTrue authentication service. Embedding role information directly into the JWT avoids additional authorization lookups during request execution and allows PostgreSQL to evaluate access decisions without application-layer coordination. This keeps role checks lightweight and avoids additional database round-trips. For regular authenticated users, access is not primarily determined by role but by ownership. Policies on the `Projects` table therefore restrict access to rows where the stored `user_id` matches the session identity:

```
create policy "Users can read own Projects" on public.Projects
  for select
  using (auth.uid() = user_id);
```

This pattern (comparing `auth.uid()` with the row's owner column) is applied consistently across all protected tables. As a result, the `user` role itself carries little semantic weight at the database level. Any authenticated session, regardless of its role value, is by default confined to its own data. Role differentiation only becomes relevant once a session needs to operate beyond its ownership scope. This is the case for administrators. Instead of defining a large number of special-case policies, a single catch-all rule grants unrestricted access:

```
create policy "admin full access" on public.Projects
  for all
```

This keeps the overall policy set compact. Ownership-based restrictions form the default, while administrative privileges are added on top. By convention, each migration that introduces a new table, should also receive such an admin policy, ensuring that the access model remains consistent and avoids the need to revisit access logic at the application level.

The `reviewer` role, in contrast, is intentionally not enforced at the database level. Reviewer access is inherently Project-specific. A reviewer should be able to access a particular Project, not the entire dataset. Such fine-grained permissions cannot be expressed through a simple JWT role check, but require a per-row mapping between users and Projects. This would typically be implemented through an invitation or access table and evaluated via joins. While the role is already defined in the schema, the implementation of the corresponding mechanism is in planning (see Section 3.3, F32). A similar consideration applies to the `user_status` enum. Although it captures the lifecycle of user accounts, it is currently not evaluated in any RLS expression. Consequently, a suspended user's session is not automatically restricted at the database level. Enforcement would instead need to occur either in the application layer or through explicit session revocation in GoTrue.

Guest access follows a slightly different approach. Instead of introducing a separate identifier model, the system relies on anonymous authentication sessions provided by the underlying supabase authentication infrastructure. When an unauthenticated user initiates a conversion, the backend provisions a temporary authenticated session with its own unique identity. From the database perspective, these anonymous users behave like regular users. They own their Projects under the same ownership model and no special-case policies are required. The session itself is stored in an `httpOnly` cookie managed by the Supabase SSR client. An additional `is_anonymous` flag in the JWT allows the application to distinguish between anonymous and registered users where necessary. If a user later decides to register, the anonymous session can be upgraded in place via `supabase.auth.updateUser()`. Because the `user_id` remains unchanged, all previously created Projects remain accessible without any form of data migration.

At the API layer, utility functions distinguish between authenticated, registered, and anonymous sessions depending on the access requirements of a route. This allows endpoints that support guest workflows to accept temporary anonymous identities, while operations requiring long-term ownership or account management remain restricted to registered users.

Independently of these route-level checks, application services perform explicit ownership verification checks before executing write operations. This introduces a second authorization boundary in addition to the database-level policies and protects against accidental misuse at the application layer.

Finally, client-side enforcement is handled through the `AuthContext`, which exposes the authenticated user's role along with convenience flags such as `isAdmin`, `isReviewer`, and `isUser`. These are used by layout components to conditionally render UI elements or redirect unauthenticated users. It should be emphasized, however, that this layer is purely presentational. It complements, but does not replace, the server-side and database-level controls described above.

5.4.2 State-Aware Access

Beyond role-based rules, access to Projects in QDArchive is further shaped by the lifecycle state of individual resources. In practice, there are three mechanisms that contribute to this. The public visibility flag on Projects, Share Links with configurable permissions, and soft deletion.

Each Project carries an `is_public` boolean, which determines whether it is accessible without authentication. Projects where `is_public = true` are exposed through a dedicated database view, `archive_projects`, which is defined with `security_invoker = on` (PostgreSQL, 2026e). This configuration is essential, as without it, PostgreSQL would evaluate queries against the view under the privileges of the view's creator rather than those of the caller, effectively bypassing any row-level security policies defined on the underlying tables. By enabling `security_invoker`, all queries are executed in the caller's security context, ensuring that RLS constraints remain active regardless of how the view is accessed. This preserves the integrity of the access control model even as the view abstraction evolves. To propagate public visibility to related entities, RLS policies on `project_versions` and `project_version_files` use `EXISTS` subqueries:

```
create policy "public versions metadata readable"
  on public.Project_versions for select
```

```
using (
  EXISTS (
    SELECT 1 FROM public.Projects p
    WHERE p.id = Project_versions.Project_id
      AND p.is_public = true
  )
);
```

This approach ensures that visibility is derived consistently from the parent Project and does not need to be stored redundantly across multiple tables. The same principle is applied to object storage. A corresponding storage policy allows unauthenticated download of files whose parent Project is public by traversing the full relationship chain (`files → project_version_files → project_versions → projects`). Files belonging to private Projects, in contrast, remain accessible only to the owner or an administrator. When a Project is made public via the update endpoint, the service layer automatically removes any existing Share Links. Since public resources no longer require token-based access, retaining such links would be redundant and could lead to stale or misleading access paths.

For private Projects, Share Links provide controlled external access. Each Project may have at most one such link. A Share Link consists of a randomly generated UUID token, which is embedded in a URL of the form `/archive/secure/{token}`, a `disabled` flag for temporary revocation, and a `permission_level` indicating the intended scope of access (`view`, `review`, or `edit`). This design keeps the access model simple while still allowing for future extensions. Link resolution is handled by a public API endpoint that does not require authentication. The method `ShareLinkService.resolveSecureLink` retrieves the corresponding record, verifies that the link is active, and returns the associated Project identifier. In the current implementation, access is granted only if the permission level is `view`. The additional levels `review` and `edit` are already defined at the schema level but are not yet enforced by differentiated logic, effectively acting as forward declarations for a more fine-grained access model. Management of Share Links (including creation, toggling, regeneration, and permission updates) is restricted to the Project owner. Each corresponding API route requires authentication and delegates to the `ShareLinkService`, which performs ownership checks before applying any changes. At the same time, RLS policies on the `share_links` table enforce the same constraint at the database level, ensuring consistency between application and persistence layers.

Soft deletion also affects effective visibility. Records with a non-null `deleted_at` value are excluded from normal application queries and therefore no longer appear in the active archive, even though they remain present in the database. Deleted resources are still subject to the same RLS policies as active records. They are hidden rather than physically removed.

Taken together, these mechanisms show that access and visibility of Projects in QDArchive is not purely role-driven. Instead, it is context-sensitive and evolves with the state of the underlying resource. A Project's effective visibility is therefore the result of multiple interacting factors, such as ownership, public status, Share Link configuration, and deletion state, rather than a single controlling attribute, as already introduced in the architectural design (see section 4.6).

5.5 Application Logic and API Design

This section describes how the system's functionality is exposed through a structured API and how application logic is organized to translate incoming requests into domain operations.

5.5.1 Resource Structure

The HTTP API follows a resource-oriented design in which each URL identifies a domain entity and the HTTP method specifies the operation to be performed on it, in line with REST architectural principles (Fielding, 2000). The resource hierarchy mirrors the aggregate structure described in Section 5.2.1. A Project is the top-level resource, versions are subordinate to Projects, and files are subordinate to versions. This hierarchical relationship is directly reflected in the URL structure:

```
/api/projects
/api/projects/{id}
/api/projects/{id}/versions
/api/projects/{id}/versions/{versionId}
/api/projects/{id}/versions/{versionId}/files
/api/projects/{id}/versions/{versionId}/files/{fileId}
/api/projects/{id}/versions/{versionId}/download
```

Standard HTTP semantics are applied consistently. `GET` retrieves a resource or collection, `POST` creates a new subordinate resource, `PATCH` applies partial updates, and `DELETE` removes a resource. Where an operation does not fit neatly into CRUD semantics, a dedicated sub-resource is used. For instance, regenerating a Share Link token, for example, is expressed as `POST /api/shareLinks/{ProjectId}/regenerate` instead of updating a field via `PUT`. This makes the intent of the operation explicit at the level of the URL itself and signals that the server, not the client, controls the generated value. A distinction that would be lost if the token were treated as an updatable field.

The nesting depth is intentionally limited to three levels, reflecting a balance between two competing concerns. On the one hand, deeper hierarchies would tightly couple clients to the internal aggregate structure and result in unwieldy URLs. On the other hand, shallower paths would remove the ownership context from the request, requiring the server to reconstruct the full resource hierarchy from the database before any authorization check can be performed. In practice, each route handler resolves the parent resource before operating on the child. A request such as `DELETE/api/Projects/{id}/versions/{versionId}/files/{fileId}` therefore first verifies Project ownership before inspecting the version or the file relation. The `ProjectId` in the path thus serves a dual purpose. It acts both as a routing parameter and as an anchor for authorization decisions.

Not all endpoints fit naturally into this hierarchy. Archive and conversion operations are intentionally placed outside of it, as they are either read-only or cross-cutting in nature:

```
/api/archive/projects               | public Project discovery
/api/archive/secure/{secureLinkId}  | share-link resolution
/api/convert                        | format conversion
/api/convert/exportFormats          | supported format enumeration
```

5.5.2 Route Handler Conventions

Next.js route handlers serve as the entry point for all API requests. Each handler receives a typed `NextRequest` object and returns a `NextResponse`, with native access to server-side cookies via `next/headers`. These cookies are used consistently across endpoints for session management and anonymous guest session handling. Dynamic route segments such as `[id]` and `[versionId]` map directly to the REST resource hierarchy without additional router configuration, keeping the URL structure self-documenting and aligned with the aggregate model described in Section 4.3.

All handlers are wrapped in the `withErrorHandling` higher-order function, which provides uniform trace identifier assignment and error mapping across the entire API surface. A middleware layer validates JWTs and enforces route-level authentication guards before requests reach handler code. This centralizes authentication enforcement and ensures consistent behavior across all protected routes.ed individually per route.

5.5.3 Request Lifecycle

Although the API surface is relatively large, all route handlers follow a common processing pattern. Figure 5.9 illustrates this flow for a representative write operation. The implementation itself is technically conventional. The architectural relevance lies primarily in three design decisions.

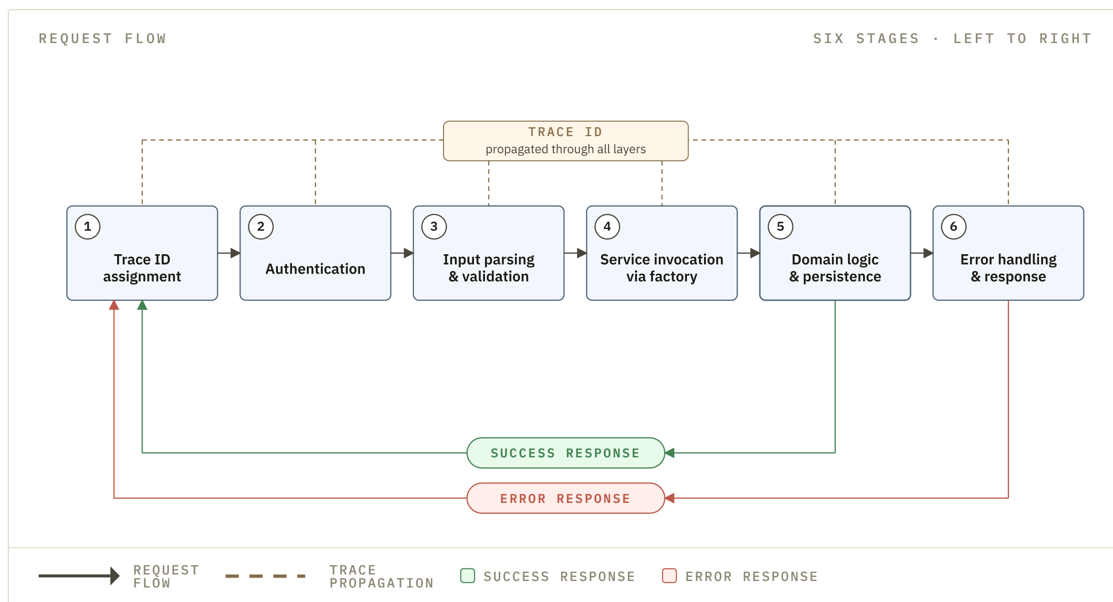

Figure 5.9: Request lifecycle from client to database interaction

Authentication is handled through three dedicated helper functions that distinguish explicitly between authenticated, registered, and anonymous access. This is handled directly at the API boundary, rather than delegating this distinction to the service layer. As a result, authorization semantics are kept consistent across endpoints and guest-session handling logic is prevented from leaking into domain services.

Services are instantiated per request through factory functions that assemble repositories and infrastructure adapters by constructor injection. This is not merely a stylistic

choice, but follows from the request-scoped authentication model of the application. A new database client is created for each incoming request and bound to the authentication and session context of the current user. Reusing service instances across requests would therefore risk executing database operations under an incorrect authentication context. By propagating the request-scoped client through the full dependency chain (factory, service, and repository) the architecture ensures that all database operations execute under the correct user identity and remain compatible with PostgreSQL row-level security policies. At the same time, dependency composition remains explicit and localized without requiring global state or a centralized dependency injection container.

Multi-step write operations are coordinated through a Saga-based compensation mechanism. If one step fails, previously completed steps are reversed in order to restore a consistent state. This approach is necessary because the Supabase JavaScript client does not expose database-level transactions, while operations such as Project and ProjectVersion creation still require recoverable multi-step consistency guarantees. Remaining non-atomic paths are partially mitigated by the append-oriented persistence model, in which incomplete writes produce isolated intermediate states while preserving the structural integrity of the underlying data. The implications of this trade-off are discussed further in Chapter 6.

Together, these three mechanisms ensure that authentication context, dependency composition, and consistency guarantees are propagated coherently throughout the request lifecycle.

5.6 Integration of External Services

Beyond the core application logic, the system relies on a combination of platform-provided and external services to implement functionality such as authentication, file storage, and format conversion. While authentication and storage are provided as part of the Supabase backend platform, the format conversion functionality is integrated as an external dependency. To prevent tight coupling, all such interactions are mediated through dedicated abstraction layers. This isolates infrastructure concerns, preserves module boundaries, and ensures that individual services can be adapted or replaced without affecting the rest of the system.

5.6.1 QDConvert

Format conversion is handled by the external library QDConvert, which encapsulates all format-specific parsing and serialization logic for CAQDAS Project files. The library treats conversion as a two-step process. Any supported input format is first parsed into a canonical in-memory representation called the corpus, and the corpus is then serialized into any supported output format. QDArchive itself never needs to understand the internal structure of individual QDA formats, it only works with this format-agnostic intermediate representation.

Although QDConvert is developed alongside QDArchive, it operates at a different level of abstraction. It reasons about file formats, conversion pipelines, and format-specific representations rather than archival artifacts or lifecycle states. Centralizing all interactions with QDConvert in the `ConvertService` keeps these concerns explicitly separate. Conversion-specific types and error models are translated into the archival domain vocabulary at a single point, which simplifies testing and prevents conversion concerns from becoming entangled with lifecycle or access-control logic.

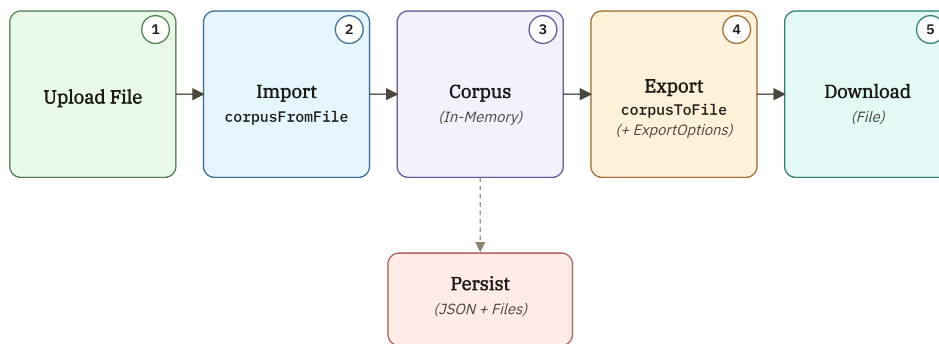

Figure 5.10: Conversion pipeline from file upload to export and persistence

As shown in Figure 5.10, a conversion request is processed in two steps. First, the uploaded file is parsed into a corpus via `corpusFromFile`. The library identifies the input format automatically from the file structure. Second, the corpus is serialized into the requested output format via `corpusToFile` and streamed back to the client. Optional export parameters (e.g. packaging of source files or OS-specific encoding) are passed through transparently. This two-step model is what makes conversion format-agnostic. Adding support for a new format requires only a new parser or serializer in the library, with no changes to QDArchive's application logic. The same first step is reused in Project creation and versioning workflows. In these cases the corpus is not returned to the client but persisted as domain state, as described below.

As shown in Figure 5.10, a conversion request is processed in two steps. First, the uploaded file is parsed into a corpus via `corpusFromFile`. The library identifies the input format automatically from the file structure. Second, the corpus is serialized into the requested output format via `corpusToFile`. The resulting file is persisted to object storage and associated with the corresponding ProjectVersion, from which it can later be retrieved through a signed download URL. Optional export parameters (e.g. packaging of source files or OS-specific encoding) are passed through transparently. This two-step model makes conversion format-agnostic. Adding support for a new format requires only a new parser or serializer in the library, without requiring changes to QDArchive's application logic. The parsing step is also reused during `ProjectVersion` creation and update workflows. When creating or updating a `ProjectVersion`, the uploaded file is transformed into the corpus representation and imported into the archival domain model instead of being serialized back into an output format. This allows the same conversion pipeline to support both format interoperability and artifact persistence.

The transformation of the imported corpus into persistent domain entities is illustrated in Figure 5.11. The mapping separates the format-agnostic representation of the analysis from the associated binary artifacts, reflecting the dual nature of qualitative data as both structured metadata and external source material.

When a conversion result is persisted, the corpus must be mapped to QDArchive's domain model. The corpus format contains two structurally different kinds of content that require different storage strategies.

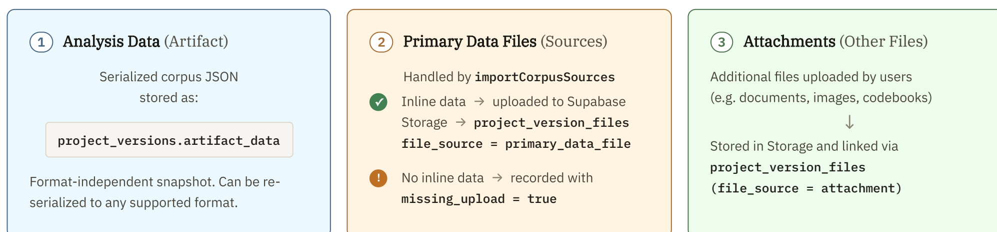

Figure 5.11: Mapping of conversion output to domain entities

The first is the corpus representation itself, which captures the analytical structure of the artifact, including codes, coded segments, hierarchical relationships, and memos. This representation is serialized and stored as `artifact_data` on `project_versions`. This snapshot enables re-export to any supported format without re-parsing the original file. The second is the set of sources, which are primary data files referenced inside the corpus, such as transcripts or recordings. These are processed by `importCorpusSources` in the `FileService`. Embedded files are uploaded to object storage and linked to the version as `primary_data_file` entries. Files that are only referenced but not included in the upload are still recorded, flagged with `missing_upload = true`. This preserves the structural completeness of the analysis even when underlying files are absent, which is a common situation with CAQDAS exports that reference external media. By retaining these references, QDArchive supports incremental completion of the artifact, allowing missing sources to be uploaded and associated with the corresponding `ProjectVersion` after the initial import.

The result is a three-part domain representation that consists of serialized analytical structure, extracted primary sources, and any additional user-provided attachments. This separation allows QDArchive to reconstruct the original research context as faithfully as possible while remaining robust to the incomplete exports typical in practice.

Unsupported output formats are rejected via `validateFormat` before any processing begins. During parsing, `qdconvert` returns a structured result object containing both a success flag and an array of parser warnings. When parsing fails, the `ConvertService` incorporates these warnings into the thrown error message to surface diagnostic context to the caller. These are handled centrally by the `withErrorHandling` wrapper, which produces a standardized HTTP response with the request's trace identifier attached.

5.6.2 Infrastructure

In addition to the conversion library, the system relies on authentication and object storage infrastructure services provided by the Supabase platform. In both cases, infrastructure-specific details are kept out of the domain layer, though the encapsulation pattern differs. Object storage is wrapped behind a dedicated `StorageService` class, while authentication is handled through a combination of typed utility functions and a focused `IdentityService`.

Authentication User authentication is provided by GoTrue, the authentication component of the self-hosted Supabase stack. Session management and identity resolution are handled through three typed utility functions (`requireUser()`, `getRequestUserId()`, and `getRegisteredUserId()`) that wrap the `@supabase/ssr` client and are invoked directly from route handlers. This client handles the details of session management in a

server-rendered Next.js context, including token refresh and cookie handling, and keeps authentication concerns out of the service layer without requiring a dedicated wrapper class for standard operations. Two client instances are maintained. A user-scoped client operates under the caller's row-level security context and is used for all request-bound operations. A separate service-role client with an unrestricted key is reserved for server-side administrative tasks such as applying migrations or seed operations. The service-role key is never exposed to the browser. Operations not covered by the SDK are centralized in the `IdentityService`, which issues direct HTTP calls to the GoTrue REST API. Password recovery, for instance, is implemented via `/auth/v1/recover` to trigger the recovery email, followed by a direct password update using the access token issued by the recovery link. Centralizing these interactions keeps the rest of the application insulated from the specifics of the authentication protocol.

Storage Binary data is managed through the `StorageService`, a thin adapter around the Supabase storage client and provides operations for file upload, signed URL generation, and download. The service itself contains no domain logic. All decisions about file ownership and lifecycle are handled in the Share module. Files are organized in a hierarchical path structure within the `qdarchive` bucket. The path reflects the ownership and versioning model of the system, grouping files first by user, then by Project, and finally by Project version. Each stored object is assigned a unique identifier while preserving the original filename.

```
users/{userId}/Projects/{ProjectId}/versions/{versionId}/{uuid}-{filename}
```

This structure ensures that files can be efficiently located and associated with their corresponding domain entities without requiring additional lookup tables. Before storage, filenames are normalized to remove non-alphanumeric characters. Access to stored files is provided via short-lived signed URLs, which include a `Content-Disposition` header to restore the original filename on the client. Uploads are limited to 200 MB, as enforced by the storage layer. By encapsulating all storage interactions in a dedicated service, the system remains independent of the underlying storage provider. Replacing the backend— for example, switching from a self-hosted MinIO instance to an S3-compatible service—would only require changes within the `StorageService`.

5.7 Deployment

The application is deployed as a set of Docker containers managed via Docker Compose. Rather than relying on a single configuration, the stack is defined across multiple Compose files that are combined at startup. These cover the application itself, the Supabase backend services, object storage, and the API gateway, allowing each concern to be configured and evolved independently.

Application container The Next.js application is packaged as a single Docker image built from the `web` directory. A multi-stage build is used to produce a lean production image while keeping build-time dependencies separate. Access to the private npm package QDConvert is handled through build-time secrets.

Supabase stack The backend services are provided by a self-hosted Supabase deployment. PostgreSQL serves as the primary database, while GoTrue handles authentication and JWT issuance. PostgREST exposes the database through a REST interface used by

the Supabase JavaScript client, and Kong acts as the internal API gateway. Supavisor provides connection pooling in transaction mode. Additional containers support object storage (backed by MinIO), image processing (Imgproxy), schema introspection (pg-meta), and infrastructure log aggregation (Vector). Realtime and Edge Functions are part of the standard Supabase stack but are not used by the application. A dedicated migration container applies pending schema changes and optionally seeds the database on startup.

Environments and Networking Development and production run side by side on the same host but are strictly isolated. Each has its own Docker network and port range, with the active environment determined by the `DEPLOY_STAGE` variable. Configuration is supplied through separate environment files excluded from version control, with template files documenting the required variables.

TLS and External Access Nginx Proxy Manager sits in front of both environments, handling TLS termination via Let's Encrypt and host-based routing. This means both instances are reachable securely through a single public endpoint without exposing individual container ports directly.

6 Evaluation

This chapter evaluates the QDArchive backend architecture against the requirements baseline defined in Chapter 3. Rather than focusing on functional correctness or raw performance, the evaluation examines whether the implemented architecture satisfies the previously identified quality attributes, in particular with respect to modularity, maintainability, and controlled access to domain resources.

The evaluation indicates that the architecture largely succeeds in enforcing clear module boundaries, isolating infrastructure concerns, and preserving a format-independent domain model through dedicated abstraction layers. At the same time, several trade-offs and limitations remain, most notably the absence of database-level transactions for multi-step operations and the intentionally simplified role model. A recurring theme throughout the evaluation is the distinction between conceptual architectural validity and the completeness of its operational realization. In several areas, intended consistency and access-control guarantees are structurally represented within the architecture but not yet enforced uniformly across all system operations. Accordingly, the focus of this chapter lies not only on whether architectural mechanisms exist in principle, but also on whether they are applied consistently enough to provide reliable guarantees under failure, concurrency, and evolving lifecycle conditions without requiring excessive architectural complexity or introducing domain inconsistencies.

6.1 Evaluation Methodology

Architectural quality cannot be assessed through quantitative benchmarks alone. Instead, software architectures must be evaluated against explicitly defined quality attributes and architectural trade-offs (Bass et al., 2021; Kazman et al., 2000). Following this approach, the evaluation focuses on how the implemented structure supports or constrains change, dependency control, consistency, and access enforcement under realistic change and evolution scenarios rather than on isolated runtime metrics.

This thesis addresses three central architectural questions: (i) how heterogeneous CAQDAS formats can be integrated into a format-agnostic archival model while preserving analytical structures, (ii) how lifecycle-aware access control can be enforced consistently across application and persistence layers, and (iii) whether the resulting architecture remains maintainable and adaptable as requirements evolve. These questions cannot be answered adequately through isolated runtime measurements alone. The quality of a system's architecture can only be assessed over time, as the system is changed and extended. A well-designed architecture enables such evolution while preserving internal coherence, maintaining clear separation of responsibilities, and preventing unintended dependencies. Quantitative measurements may indicate the performance characteristics of a specific deployment configuration. However, they provide limited insight into whether format-specific volatility remains isolated from the core domain model or whether publication workflows can evolve without introducing inconsistencies across enforcement layers.

The evaluation therefore follows a qualitative, scenario-based approach. Module boundaries are examined against the intended dependency structure described in the architecture chapter, access control is traced from policy definition to enforcement across both the application and database layers, and hypothetical change scenarios are used to assess how modifications propagate through the system architecture. Where possible, these obser-

vations are grounded in concrete implementation details such as RLS policies or service interfaces, which serve as empirical reference points for the analysis.

The evaluation criteria are derived from the ISO/IEC 25010 quality model introduced in Chapter 3. The analysis focuses primarily on the quality attributes most directly shaped by the systems architecture. It examines the system's maintainability, its handling of security-relevant data and operations, its behavior under increasing load, its operational reliability, and its ability to integrate with other systems. Some aspects are discussed only briefly. Ease of use and accessibility are shaped mainly by the surrounding platform and user interface. Privacy and deployment also depend on factors beyond the backend architecture. For this reason, they are considered where relevant.

One limitation of the present approach should be acknowledged explicitly. The evaluation is conducted by the same person who participated in designing and implementing the system. While this provides detailed internal knowledge of the architecture and its trade-offs, it also introduces a potential risk of confirmation bias. To mitigate this, claims are grounded in observable structural evidence wherever possible, and limitations or partial realizations of the intended design are discussed throughout the evaluation.

6.2 Requirement Coverage Analysis

The architecture is evaluated against the functional and non-functional requirements defined in Sections 3.3 and 3.4. Rather than assessing each requirement individually, the analysis groups related requirements into functional capabilities that share a common architectural realization. This aggregation reflects the fact that architectural decisions operate at the level of modules, data structures, and interaction patterns, which typically support multiple requirements simultaneously.

Table 6.3 maps these aggregated functional capabilities to the architectural mechanisms responsible for their implementation and indicates the degree of coverage achieved in the current system. The grouping follows the three solution areas introduced earlier (Find, Convert, Share) and clusters requirements that are realized through the same structural components (e.g. F1–F6 for Find, F22–F27 for Project and version management).

Capability (FR groups)	Architectural Mechanism	Status
Search and public discovery (F1–F6)	`archive_projects` view, generated `tsvector` column, GIN indices, Find module	Fully satisfied
Format conversion (F7–F16)	ConvertService translation boundary, QDConvert integration, session-based Project creation	Fully satisfied
Project and version management (F22–F23)	Project/Project Version aggregates, repository layer, copy-on-write versioning model	Fully satisfied
File storage and retrieval (F24–F27)	Supabase Storage, Storage Service abstraction, signed URL access	Fully satisfied
Access control and sharing (F28–F30)	PostgreSQL RLS, Share Link Service, UUID token resolution	Partially satisfied
User management (F17–F21)	GoTrue integration, Identity Service, `user_status` enum	Partially satisfied
Administrative workflow (F31–F32)	Architectural model only (review queue not implemented)	Not implemented
Background services (F33–F35)	Not realized in implementation scope	Not implemented
Soft deletion and audit trail (F26–F27)	`deleted_at`/`deleted_by` across domain tables	Fully satisfied

Table 6.3: Aggregated mapping of functional requirements (Section 3.3) to architectural mechanisms and implementation status

Table 6.4 maps the quality attributes from Section 3.4 to the structural mechanisms that realize them. While the functional mapping focuses on feature-level coverage, the quality attribute mapping evaluates the architectural properties that cut across the entire system.

Quality Attribute	Architectural Mechanism	Status
Maintainability	Modular structure, explicit module boundaries, conversion service as as translation boundary, factory-based composition	Fully satisfied
Security	RLS and service-layer checks, JWT-based identity, Supabase anonymous authentication, secure share tokens, server-side service-role key	Partially satisfied
Scalability	Indexed query paths (B-tree, GIN), stateless application layer, Supabase connection pooling	Partially satisfied
Interoperability	Format-agnostic artifact representation, translation boundary between external CAQDAS formats and the Share model, isolation of format-specific semantics, REFI-QDA support	Fully satisfied
Reliability	Trace identifiers, structured error handling, per-item batch reporting, Saga-based compensation for critical write operations, soft deletion	Partially satisfied
Usability and accessibility	Anonymous sessions, public search, conversion without registration	Fully satisfied
Privacy and compliance	Terms of Service enforcement, HTTPS-only communication, user-controlled profile data	Fully satisfied
Distribution and deployability	Environment-based configuration, containerized deployment, interface-based external services	Fully satisfied

Table 6.4: Mapping of quality attributes (Section 3.4) to architectural mechanisms and implementation status

Fully satisfied capabilities Several core capabilities are fully realized, as the corresponding architectural mechanisms directly enforce the required behavior. Search and public discovery satisfy requirements F1–F6 through a combination of full-text search capabilities, optimized indexing strategies, and a dedicated read model that exposes only publicly visible artifacts while preserving database-level access control. As a result, both efficient retrieval and access isolation are enforced at the database level instead of relying solely on application-side filtering. Format conversion is similarly isolated through the conversion service, which encapsulates all interaction with the external conversion library behind a stable internal representation. This representation acts as a translation boundary between heterogeneous CAQDAS formats and the archival domain model by mapping format-specific structures onto a common archival representation. As a result, vendor-specific semantics and format complexity remain confined to the conversion layer and do not propagate throughout the remainder of the system. Project and version man-

agement are supported through aggregate-oriented services and a copy-on-write versioning model that preserves historical consistency without mutating existing state. Storage and file access are separated cleanly through the combination of metadata entities and signed object-storage URLs, reducing coupling between domain logic and binary file handling. Quality attributes that are less directly shaped by the backend architecture, such as interoperability, usability, privacy, and deployability, can (to a large extent) be considered fulfilled through the mechanisms discussed in the methodology section. Therefore these are not examined any further here.

Partially satisfied capabilities Some aspects of access control and lifecycle management remain incomplete. Ownership-based access and administrative privileges are enforced, but role-specific behavior (notably the Reviewer role) is not yet realized in either RLS policies or service logic. Similarly, the `user_status` field is not evaluated at runtime, meaning that suspended users retain full access as long as their JWT remains valid. Share Links follow a comparable pattern: the underlying mechanism is in place, but only the View permission level is currently enforced.

Reliability is partially addressed through a Saga-based compensation mechanism for critical multi-step operations such as Project and version creation. Should any step fail, previously completed steps are reversed, restoring both database records and any already-uploaded storage objects to a consistent state. However, not all multi-step operations are covered. The sequence of making a Project public and deleting its Share Link, for example, remains non-atomic. The append-oriented data model provides an additional safety net. Since operations only add records and rely on soft deletion rather than in-place updates, incomplete writes result in isolated intermediate states rather than structural corruption.

Scalability is currently limited to vertical scaling. No mechanism for distributing conversion workloads across multiple instances has been implemented, and horizontal scaling behavior has not been empirically validated.

Not yet realized capabilities Some functional requirements defined in Chapter 3 are not yet reflected in the operational behavior of the system. Administrative workflows (F31–F32), including the review queue for externally sourced projects, exist as architectural concepts in the domain model but have no corresponding implementation. Background services (F33–F35), covering external discovery, change monitoring, and scheduled backup, are likewise defined in the specification but require scheduler infrastructure that falls outside the current implementation scope. As a result, a developer inspecting the domain model could reasonably assume behaviors, such as automated lifecycle processing, role-based reviewer access, or scheduled ingestion, that are not currently guaranteed by the implementation. These gaps represent not only missing functionality, but also a divergence between the conceptual model and the operational system. The architectural foundations for addressing them are in place and their realization is treated as a priority in Chapter 7.

6.3 Scenario Validation

Chapter 3 defines five quality attribute scenarios with explicit response measures. This section closes each scenario by assessing whether the implemented architecture satisfies the stated measure, and identifies any conditions under which the measure is only partially met.

6.3.1 Modifiability Scenario (Adding a New QDA Format)

Response measure: Zero modifications to the Project, versioning, file management, or access control subsystems.

The architecture fulfills this scenario to a large extent. As demonstrated in the library replacement analysis in Section 6.7, all interaction with the conversion library is encapsulated behind the conversion service. Domain code outside this boundary only depends on the internal corpus representation and the service's own typed interface. Adding a format that the conversion library supports requires no changes to QDArchive application code. Not even the conversion service requires modification, since format discovery, validation, and parsing are delegated entirely to the library at runtime. Consequently, a dependency version update is sufficient.

6.3.2 Security Scenario: Share Link Enumeration

Response measure: The token space provides sufficient entropy that the probability of a correct guess within a practical number of attempts is negligible.

The implementation fulfills this scenario. Share Link tokens are generated as UUID v4 values, providing 122 bits of effective entropy (Leach et al., 2005). At a rate of one billion guesses per second, exhausting the space would take longer than the estimated age of the universe. Thus, the response measure is met.

6.3.3 Security Scenario: Privilege Isolation (Application-Layer Bug Bypasses Ownership Check)

Response measure: The database returns an authorization error and no records belonging to other users are exposed.

This scenario can be considered satisfied with qualification. PostgreSQL's RLS policies enforce ownership and visibility constraints independently of application logic. A query executed under a user's session cannot retrieve rows it does not own, regardless of whether the route handler performed an ownership check. The qualification is that this guarantee holds only for resources currently covered by RLS policies. The `reviewer` role and the `user_status` suspension mechanism are defined in the schema but not yet enforced by any RLS policy. For these cases, an application-layer bug that skips an ownership check has no database-level backstop. The scenario's response measure is met for the currently enforced access rules and not met for the as-yet-unenforced ones.

6.3.4 Access Control Consistency Scenario (Share Link Invalidation)

Response measure: The old Share Link returns an error and the Project appears in public search results.

This scenario is partially satisfied. The application layer implements the required behavior. When a Project is set to public, the active Share Link is deleted and the visibility flag is updated. The Project subsequently appears in the public archive view and becomes discoverable via search. However, the two operations execute as sequential independent database calls without a Saga-based compensation mechanism. A failure between the two steps leaves the Project public with its Share Link still active. This means the response measure is met under normal operation but not guaranteed under partial failure. Ex-

tending the Saga pattern to cover this transition, identified in Chapter 7, would close the remaining gap.

6.3.5 Reliability Scenario

Response measure: No orphaned records or corrupted version entries remain after a failed multi-step operation.

This scenario is fulfilled for the critical multi-step operations, but not universally. For the two operations most likely to produce orphaned records (Project and Projectversion creation) a Saga-based compensation mechanism reverses completed steps when a subsequent step fails. If a file upload succeeds but the database write fails, the storage objects are deleted. If the version record is created but a subsequent file metadata write fails, the version record is rolled back. However, the current implementation assumes that compensation itself succeeds reliably. Compensation failures are not surfaced explicitly and may therefore leave orphaned external state undetected. The architecture thus establishes a clear rollback model conceptually, but does not yet fully guarantee operational observability of incomplete recovery paths. The response measure is met for these operations under normal failure conditions. For lower-risk multi-step sequences not yet covered by the Saga pattern, the append-oriented data model provides a partial safety net. Incomplete writes produce isolated intermediate states rather than structural corruption. However, the response measure as stated, which is that no orphaned records after any failed operation remain in storage, is not universally met.

6.3.6 Anonymous Session Upgrade Scenario

Response measure: All Projects created during the anonymous session are retrievable under the registered user's identity without any data migration step.

This scenario, in which a guest user registers after a successful conversion job while retaining access to previously created Projects, is satisfied by the current implementation.Anonymous sessions are issued as standard authentication tokens, and guest projects are owned by the anonymous user's identifier, which is the same identifier used by access control policies for all users. When the user registers, the anonymous session is promoted to a permanent identity without changing the underlying identifier. Project ownership is therefore preserved without any data migration step, and all previously created Projects remain accessible under the registered identity.

6.4 Modularity and Coupling Analysis

The modular monolithic architecture aims at maximizing cohesion within modules by grouping related responsibilities, while restricting dependencies between modules to explicitly defined and controlled interaction points. This section examines how well these principles are realized in the current implementation. In addition to module boundaries, the analysis also considers how the domain model and its aggregate structure are realized in the implementation, with particular attention to the extent to which aggregate behavior is implemented in the service layer rather than encapsulated within domain objects.

6.4.1 Cohesion Within Modules

The core Share module is internally structured into the three submodules `Project`, `ProjectVersion`, and `File`. Each is following a consistent layered pattern of service,

repository, and factory. Responsibilities are clearly delineated. Project-level operations are handled in the `ProjectService`, versioning of Projects and copy semantics in the `ProjectVersionService`, and file management (including uploads and associations) in the `FileService`. This separation results in a high degree of cohesion. Changes to file handling, for instance, are largely confined to `FileService` and its repository without propagating into Project or versioning logic. The structure remains predictable across submodules, which reduces cognitive overhead and supports local reasoning about changes. This, in turn, directly contributes to the maintainability of the system, as modifications can be performed within well-defined boundaries without unintended side effects.

A comparable level of cohesion is achieved in the Convert module. All interaction with the external QDConvert library is concentrated in the `ConvertService`, which acts as a dedicated translation boundary. The service encapsulates the interaction with the library and exposes its functionality through a stable internal interface, shielding the rest of the system from library-specific APIs and implementation details. This creates a stable boundary. Replacing or extending the conversion library would primarily affect this single component, without requiring changes in the Share module. The interaction between the Share and Convert modules introduces the most relevant inter-module dependency in the current implementation. The Share module relies on the `ConvertService` for corpus parsing and serialization during artifact creation and versioning, while the Convert module itself remains independent from archival lifecycle management and domain state. Although this introduces a dependency from the core domain to a supporting module, the coupling remains narrow and explicitly bounded through the conversion service interface. No circular dependencies between modules are introduced.

The Find module is deliberately isolated. It exposes read-only access to public artifacts via the `archive_Projects` view and maintains no mutable state. Because it does not depend on internal service logic, changes to the artifact lifecycle do not propagate into the search functionality. The database view acts as a stable Projection that decouples read access from write operations, resulting in a high degree of isolation.

6.4.2 Realization of the Aggregate Model

Conceptually, the domain model treats the Project (which is in the theoretical context of QDA referred to as the artifact) as the aggregate root that defines the consistency boundary for artifact-related operations (Evans, 2004). In the implementation, however, this principle is realized in a service-oriented manner. Instead of encapsulating behavior within rich domain objects, domain logic is primarily implemented in application services (`ProjectService`, `ProjectVersionService`, and `FileService`) which coordinate validation and persistence. As a result, invariants are enforced procedurally rather than structurally. That is, correctness depends on explicitly executing a sequence of checks and operations within the service layer. For example, ownership is verified before write operations are performed, file relations are only created within the context of a version, and version creation follows a copy-on-write strategy instead of mutating existing data. While these rules are applied consistently, their enforcement depends on the correct use of service methods. They are not inherently guaranteed by the domain model itself. Consequently, consistency relies on disciplined use of the service layer instead of being intrinsically enforced by the aggregate structure. This approach simplifies the implementation and improves readability, but it partially weakens the strict encapsulation typically associated with aggregate roots (Vernon, 2016).

From a Domain-Driven Design perspective, this implementation style closely resembles what can be described as an Anemic Domain Model, in which domain objects primarily act as data containers while business rules and invariant enforcement are implemented in service classes instead of being encapsulated within the aggregates themselves (Fowler, 2003). In QDArchive, entities such as `Project` and `ProjectVersion` are represented as TypeScript interfaces without domain behavior, while coordination logic resides primarily in the corresponding application services. This approach is pragmatic and common in service-oriented web applications, particularly where workflows are dominated by persistence coordination rather than complex domain computation. However, it also reinforces the dependence on procedural correctness across service boundaries, since invariants are not intrinsically enforced by the domain objects themselves. As a result, the invariants are less explicit and potentially easier to bypass if additional code paths are introduced outside the established service layer. The evaluation identified several cases in which invariants were enforced correctly in one execution path but omitted in another, particularly in file-association and deletion workflows. This illustrates the central trade-off of the current approach. Procedural enforcement keeps the implementation comparatively simple, but depends on all service paths applying the same coordination logic consistently.

Similar trade-offs can be observed at the architectural level. Module boundaries are currently maintained through structure and development conventions rather than automated enforcement mechanisms. Although no significant cross-module dependency violations were identified during the evaluation, introducing architectural fitness functions or import-boundary checks could further strengthen long-term maintainability as the codebase evolves (Ford et al., 2022).

6.4.3 Inter-Module Communication

Communication between modules is implemented using synchronous service calls. This approach is well-suited for a modular monolith, as it reduces complexity and allows direct interaction between components within the same runtime. However, synchronous communication introduces temporal coupling. Operations that involve multiple modules require all participating components to be available at execution time. From a Domain-Driven Design perspective, asynchronous communication via domain events would provide a more decoupled and resilient integration model, allowing modules such as Convert, Share, and Find to evolve more independently (Vernon, 2013). Given the current system scope, synchronous communication represents a pragmatic and appropriate choice, but it should be noted that it also limits future decoupling.

6.5 Scalability and Performance Considerations

Although implemented as a modular monolith, the system must handle concurrent artifact processing and file operations without excessive resource consumption. This section evaluates where the current design scales effectively and where structural limitations become apparent. This directly relates to the scalability requirements defined in Section 3.4.

6.5.1 Vertical Scaling and Concurrency

All modules execute within a single runtime, which means that scalability is primarily vertical. Increasing CPU and memory resources on a single host improves throughput, whereas distributing load across multiple hosts requires additional infrastructure that is not yet in place. For the majority of operations, this is not a critical limitation. The

Node.js event-driven I/O model handles concurrent database queries and storage interactions efficiently, and most API endpoints are I/O-bound rather than CPU-intensive. Format conversion constitutes a notable exception. The QDConvert library executes synchronously within the request lifecycle. It parses the input file, transforms it into the internal corpus representation, and serializes the output without yielding control to the event loop. Unlike asynchronous I/O, this CPU-bound work blocks the event loop and cannot be interleaved with other requests (Node.js, 2026b). Under sustained load, concurrent conversion requests therefore degrade overall system responsiveness, affecting unrelated endpoints as well. A further implication is that the entire conversion pipeline (file upload, parsing, transformation, Project creation, and persistence) is executed within a single HTTP request. For large artifacts, this increases the risk of timeouts at the gateway level. Addressing this limitation would require a shift towards asynchronous processing, e.g., by introducing background jobs or chunked upload and processing strategies. This represents one important scalability limitation of the current system.

6.5.2 Database Access Patterns

All persistent state is centralized in PostgreSQL, making the database the primary scalability bottleneck. Read-heavy workloads such as public artifact discovery are optimized through targeted indexing. Partial indices on timestamp and source type columns, filtered to public projects, support efficient time-ordered and filtered queries over the public archive without scanning private rows. An array index on the tags column enables tag-based filtering, and a generated full-text search column with a corresponding index supports keyword queries across project names and descriptions. Additional indices on the soft-delete timestamp column across domain tables accelerate filtering of deleted records in repository queries. The tag-filtering and full-text search functionality are supported by GIN indices. GIN indices are well suited for multi-valued and text-search workloads, but incur higher write overhead than B-tree indices and maintain a pending-entry list that is merged synchronously once a size threshold is reached. Under sustained write load, this can lead to latency spikes during index maintenance (PostgreSQL, 2026b). While unlikely to be problematic at the current scale, this becomes relevant as ingestion volume increases.

Public queries are executed through a dedicated view configured to evaluate row-level security policies for every request, preventing unintended data exposure. The trade-off is that the query planner must account for per-row security predicates, which can prevent index-only scans on the underlying tables.(PostgreSQL, 2026f). Connection management is handled by Supavisor in transaction pooling mode, with a pool size of 20 and a maximum of 100 client connections. This configuration supports a high number of concurrent requests, as connections are returned to the pool immediately after each transaction. However, it also implies that session-level state, e.g., advisory locks or temporary configuration, is not preserved across queries. While the current system does not depend on such features, this constraint would only become relevant if more advanced concurrency control mechanisms were introduced.

6.5.3 Memory-Intensive File Processing

File uploads are currently processed using a request-parsing model that buffers the entire request body in memory before it becomes accessible to the application. As a result, there is no opportunity to stream data directly to object storage. With an upload limit of 200 MiB, memory usage scales linearly with the number of concurrent uploads. Under simultaneous large uploads, this can lead to significant memory pressure. This behavior

is not a matter of configuration but a consequence of the chosen request handling model. Mitigating it would require a different approach, such as streaming uploads directly to storage via a proxy or pre-signed upload URLs. A comparable limitation exists during archive generation. Files are retrieved from storage and buffered in memory before being added to the archive stream, causing memory consumption to scale with both file size and the number of files included in a version. Together, the upload and download paths illustrate that buffering constraints are structural characteristics of the current request-processing model rather than isolated implementation decisions.

6.5.4 Horizontal Scaling

From an architectural perspective, horizontal scaling of the application tier is feasible. The system is largely stateless. Domain data is stored in PostgreSQL and object storage, while authentication is based on stateless JWT tokens issued by Supabase Auth. No application-specific session state is maintained in memory. Any application replica can therefore handle incoming requests without relying on sticky sessions or in-memory user state. Authentication itself is delegated to the shared GoTrue instance used by Supabase, which represents an additional infrastructure dependency under horizontal scaling. In principle, multiple application replicas could therefore handle requests behind a load balancer without modification to the codebase. However, the current deployment does not include load balancing for the application container. From an architectural perspective, the benefits of such scaling would also be limited. All write operations ultimately converge on a single PostgreSQL instance. As a result, database throughput remains the limiting factor for system scalability.

6.6 Security Analysis

Security in QDArchive focuses on enforcing controlled access to research data across private, shared, and public contexts. As outlined in Section 3.4, this includes both preventing unauthorized access and ensuring consistent enforcement across application and database layers. This section examines how these requirements are realized in the current implementation, with particular attention to the layered access control model, the security of share-based access, and remaining attack surfaces.

6.6.1 Authorization Enforcement Points

Access control is enforced across three distinct layers, each operating independently. At the API boundary, route handlers validate the caller's session using dedicated authentication utilities. There are dedicated authentication utilities that provide distinct validation strategies depending on the requirements of the endpoint. The first utility accepts both registered and anonymous sessions while rejecting unauthenticated requests. The second utility explicitly restricts access to registered users, and the third accepts any session while distinguishing the absence of authentication. This design forces each endpoint to make an explicit decision about the class of caller it supports. Beyond this initial check, the service layer enforces ownership constraints before executing any write operation. Finally, PostgreSQL's RLS policies restrict which rows may be read or modified at the persistence layer. Because these policies are evaluated independently of the application logic, they act as a final safeguard against unauthorized access. Together, the three layers form a model that ensures that a failure in one layer does not automatically result in unauthorized access.

Authorization decisions are not static but depend on the lifecycle state of a Project. The same resource may transition between private, shared, and public visibility. The layered design is particularly valuable at these transition points, as it reduces the risk that partial updates leave the system in an inconsistent access state.

A more structural limitation concerns the consistency of the defence-in-depth model itself. While ownership validation is generally enforced both in the service layer and through RLS policies, this enforcement is not applied uniformly across all resource interaction paths. In particular, some file-related operations validate access to the surrounding Project or versions of Projects but do not consistently verify that the accessed resource belongs to the validated aggregate. As a result, the layered access-control model remains conceptually sound but operationally incomplete. The presence of multiple enforcement layers does not automatically guarantee that all resource relationships are validated consistently across execution paths.

A similar divergence exists between the conceptual storage access model and its runtime realization. Some storage-related RLS policies are structurally disconnected from the actual storage path conventions used by the application. Although this does not currently expose unrestricted access, it illustrates a broader architectural risk in which schema-level security assumptions may appear correct while remaining partially inactive in practice due to inconsistencies between policy definitions and operational resource organization.

6.6.2 Share Link Security

Share Links are implemented as UUID v4 tokens, providing 122 bits of entropy. Brute-force enumeration is therefore infeasible in practice. The schema enforces a one-to-one relationship between Projects and Share Links, preventing multiple active tokens for the same resource. Access to underlying files is decoupled from the token itself. Files are served via short-lived signed URLs with a 60-second expiry, meaning possession of a Share Link is not sufficient to download data directly. The client must request a signed URL through the application layer, where share permissions are evaluated before access is granted. This separation reduces the impact of token leakage and limits the time window for misuse.

6.6.3 Potential Attack Vectors

The following issues primarily reflect partial implementation of already modelled security concepts rather than fundamental architectural weaknesses. The relevant extension points and domain structures are already present, but some enforcement mechanisms remain incomplete in the current implementation state. The most prominent concerns the Share Link permission model. Although multiple permission levels are defined by the data model, only read-only access is currently supported by the permission resolution logic. As a result, links configured with elevated permissions cannot yet be used as intended. The system therefore defines a richer permission model than it enforces, which may cause confusion during future development when these levels are expected to grant differentiated access. A related gap exists in user lifecycle enforcement. While the domain model supports account suspension, the current authorization model does not yet consistently enforce this state for already authenticated users. Consequently, suspended accounts may retain access until existing sessions expire. Full enforcement would require explicit integration between account status changes and session management.

A further issue arises from the lack of transactional boundaries around the public visibility transition. When a Project is made public, the publication state transition and the deletion

of any existing Share Link execute as two sequential operations without a rollback path. Unlike Project creation and versioning, which are protected by a Saga-based compensation mechanism, this operation has no equivalent safeguard. If a failure occurs between the two steps, the system enters an inconsistent access state in which both access mechanisms coexist. While the practical impact is limited, as public visibility already grants access, the inconsistency remains undetected until corrected manually.

Finally, the absence of rate limiting at the application level introduces an additional attack surface. Computationally expensive operations, in particular format conversion, can be triggered without restriction. Under adversarial load, this could degrade system availability and disproportionately affect unrelated requests, because conversion tasks execute synchronously within the application process.

6.6.4 Guest Session Security

Anonymous access is implemented through the anonymous authentication capabilities provided by the underlying Supabase infrastructure. Each guest receives a standard authenticated session represented by a JWT stored in an HTTP-only cookie. From the database perspective, anonymous users behave like regular users. They own their Projects and are subject to the same ownership-based RLS policies, avoiding the need for special-case handling. Sessions can later be upgraded to registered accounts without changing the underlying identity reference, preserving access to previously created Projects without requiring any data migration.

The approach has two practical limitations. Anonymous sessions are tied to browser cookies. Clearing cookies or switching devices results in permanent loss of access, with no recovery mechanism for orphaned Projects. In addition, expired anonymous sessions and their associated data are not cleaned up automatically, leading to gradual accumulation of unused records without a periodic cleanup job.

6.6.5 Trust Assumptions

The security model depends on two critical components. The service-role key bypasses all RLS policies and grants unrestricted database access. Its compromise would undermine all access control guarantees. The GoTrue JWT signing secret is equally sensitive as knowledge of it allows arbitrary JWT forgery. Both secrets are stored exclusively in server-side environment variables, never exposed to the client, and excluded from the repository. The overall security of the system therefore depends not only on the application design but also on the operational handling of these credentials.

6.7 Maintainability and Extensibility

Maintainability is assessed through both scenario-based evaluation and structural analysis. First, two hypothetical change scenarios directly evaluate the quality attribute scenarios defined in Section 3.5. Format extension tests whether the adapter boundary around the conversion library holds under modification, and access control extension tests whether the distributed enforcement model remains consistent when new visibility states are introduced. Second, a structural observation examines schema evolution as a maintenance concern that neither scenario captures. It evaluates how well the system accommodates changes to the underlying data model over time.

6.7.1 Format Extension

Adding support for a new CAQDAS format primarily requires extending the QDConvert library. Because QDArchive interacts exclusively with the format-agnostic corpus representation exposed by the Convert context, no changes to the Share context, persistence model, lifecycle management, or access-control mechanisms are required. This indicates that the adapter boundary around the conversion library successfully isolates format-specific concerns from the remainder of the system.

6.7.2 Publication and Access-Control Extension

Consider introducing a new visibility state, e.g., an institutional-access level granting read permissions to members of a specific organization. This change would require updates to the persistence model, access-control rules, and sharing services. The scope of change is well-bounded, however, as all affected components live within the Share module and its supporting infrastructure. Even so, it is non-trivial and requires consistent updates across database, service, and API layers simultaneously. The absence of a centralized access control policy object means that the logic is distributed across RLS policies, SQL expressions, and TypeScript service code, making it easy to miss an enforcement point. The evaluation identified several examples in which conceptually central invariants were enforced inconsistently across different execution paths, reinforcing that distributed policy enforcement increases the long-term maintenance burden as lifecycle semantics evolve. This is the primary maintainability risk of the current access control design. Changes to visibility semantics require discipline across multiple layers rather than a single point of modification.

6.7.3 Schema Evolution

Unlike the two scenarios above, which test feature-level extensibility, schema evolution concerns the system's ability to accommodate structural changes to the data model over time. Database migrations are applied forward-only, with no automated rollback path for failed migrations in the production environment. Schema changes that affect RLS policies require the policies to be dropped and recreated in the same migration, which introduces a window during which access control guarantees may not hold. The soft-delete pattern reduces the risk of data loss during schema changes, but the accumulation of soft-deleted rows without a cleanup process represents a growing maintenance burden as the archive scales. Despite these limitations, the overall schema design remains comparatively resilient to change. Most schema changes remain localized to the persistence layer and can be introduced without affecting the higher-level domain structure.

6.8 Overall Assessment

The evaluation demonstrates that the modular monolithic architecture largely satisfies the functional and non-functional requirements established in Chapter 3. Versioning of Projects, format conversion, lifecycle-aware sharing, and access-controlled publication, which are the core capabilities of the system, are fully realized within the current implementation scope.

The identified limitations primarily reflect the current maturity level of the system rather than contradictions in the architectural model itself. They also differ in nature. Some, most notably the synchronous processing model and the reliance on a single database

instance, are conscious consequences of the current architectural priorities and the decision to favor operational simplicity over distributed complexity. Others, such as the currently unenforced Reviewer role, and the partial implementation of sharing permissions, are due to the incomplete realization of mechanisms that are already structurally anticipated by the architecture. Addressing the first category would require architectural extensions, whereas the second can largely be resolved within the existing structural model. These areas are discussed further in Chapter 7.

Against this background, the following sections revisit the three research questions introduced in Chapter 1 and assess to what extent the implemented architecture satisfies the intended design goals.

RQ1: How can analytical structures in heterogeneous CAQDAS formats be preserved through a format-agnostic archival architecture?

The evaluation indicates that the separation between archival structure and format-specific conversion logic provides an effective basis for preserving heterogeneous CAQDAS artifacts. Imported artifacts are translated into a stable internal representation that serves as the canonical form for lifecycle management, access control, and versioning. By isolating parsing and transformation concerns within the conversion layer, the architecture decouples the persistence model from vendor-specific data structures and enables consistent handling of artifacts originating from different CAQDAS environments. As a result, support for additional formats can be introduced by extending the conversion library without requiring changes to the surrounding domain architecture. At the same time, the evaluation also highlights the limits of this approach. Interoperability and semantic preservation remain partially dependent on the capabilities and abstractions provided by the external conversion library, meaning that certain format-specific semantics may still be reduced or lost during transformation.

RQ2: How can lifecycle-aware access control be integrated into the domain model without compromising architectural modularity?

The evaluation shows that access control can be integrated directly into the domain architecture by deriving permissions from artifact lifecycle state rather than from static user roles alone. Ownership, publication visibility, and Share Link configuration are enforced across two independent layers. These are on database-level row-level security policies and service-layer validation. This dual-layer approach ensures that visibility constraints remain independent of application-layer correctness while preserving clear module boundaries. Time-dependent publication semantics such as embargo periods and scheduled publication are not yet fully implemented, but the current lifecycle model provides a structurally suitable foundation for their future integration.

RQ3: To what extent does the resulting architecture preserve domain consistency while supporting extensibility, maintainability, and architectural evolution?

The evaluation demonstrates strong modularity and extensibility characteristics. The bounded-context decomposition isolates responsibilities effectively, while interoperability-related change pressure remains localized within the Convert module through the separation between archival and format-specific conversion concerns. Maintainability is supported through the layered internal structure and the factory-based service composition model, which enabled isolated testing of the primary service classes. The architecture's

primary weaknesses lie in the absence of database-level transaction support and the reliance on convention-based enforcement of certain module boundaries. Nevertheless, these limitations remain manageable within the current operational scope and do not undermine the overall consistency of the domain model.

The remaining limitations therefore primarily identify areas for future improvement and do not require fundamental restructuring of the system. Chapter 7 outlines how the existing architecture could be extended to strengthen consistency guarantees, expand lifecycle semantics, and support more advanced operational requirements as the system evolves.

7 Next Steps

This chapter outlines the most significant directions for future development, organized by architectural category rather than by feature area. These three groups of open items differ not just in scope but in kind. Some require only additional implementation within the existing module boundaries. Others depend on infrastructure that does not yet exist. A third group would change the decomposition of the system itself. Treating them separately makes clear which items are incremental and which carry architectural consequences.

7.1 Closing the Gap Between Model and Runtime

Several concepts are fully defined in the domain model and schema but are not yet enforced at runtime. These items do not require new modules or infrastructure. They require implementation within the existing boundaries. Together, they represent the gap between the conceptual system (the architecture as specified) and the operational system the architecture as enforced.

Role enforcement The Reviewer role is defined in the schema but is not evaluated in any RLS policy or service-level check. Completing the access control model requires adding RLS policies that grant reviewers read access to Projects they have been explicitly invited to, and defining the corresponding service-layer checks in the `ShareLinkService`. The `user_status` field similarly requires enforcement. Suspended or removed users should have their sessions revoked through the GoTrue administration API rather than retaining access until their JWT expires. Both gaps have well-defined insertion points in the existing module structure.

Share Link permission levels The `permission_level` enum on Share Links defines `view`, `review`, and `edit`, but only `view` is enforced during link resolution. Links with other permission levels currently raise a runtime error rather than granting differentiated access. Completing the model requires defining what operations each level authorizes and enforcing these distinctions in the route handlers and service layer that handle shared-link access.

Atomic make-public transition The sequence of making a Project public and deleting its Share Link executes as two independent database calls without a compensating mechanism. A failure between the two steps leaves the system in an inconsistent access state. The Saga pattern already applied to Project and version creation can be extended to this operation without architectural change.

Operational observability of compensation failures The current Saga implementation assumes that compensating actions succeed reliably once triggered. Compensation failures are not yet surfaced through structured logging, retry handling, or administrative monitoring, which means that orphaned external state may remain undetected under partial recovery failure. Strengthening operational observability therefore requires introducing explicit compensation tracing, failure reporting, and potentially idempotent retry mechanisms for external storage operations.

7.2 Operational Extensions

A second group of ”next steps” requires infrastructure that does not yet exist within the system. Introducing this infrastructure does not change the module boundaries or the domain model, but it opens up capabilities that currently cannot be realized.

Job scheduler as shared infrastructure Three distinct capabilities in the current requirements share a common dependency. A job scheduler capable of running periodic or time-triggered background tasks. Embargo and time-based access control require both a scheduler to evaluate pending publication schedule entries and transition Project visibility states at the defined time. Background discovery services (external repository scanning, change detection, and backups) require periodic execution outside the request lifecycle. Soft-delete cleanup requires a retention-based purge job to prevent unbounded accumulation of deleted records in the database and storage bucket. Rather than treating these as three separate features, the architectural priority is introducing a single job scheduler, e.g., via a database-backed queue table or a lightweight worker process, that all three services (external repository scanning, change detection, and backups) can use. The Convert module's current synchronous execution model could also migrate onto this infrastructure. Conversion jobs would be enqueued rather than executed within the HTTP request, and the client would poll a status endpoint or receive a notification on completion. This would resolve the primary scalability limitation identified in Chapter 6.

A related improvement concerns the current buffering model for large file uploads and archive generation. Introducing streaming-based upload and download pipelines, e.g., through direct object-storage uploads or streamed archive construction, would reduce memory pressure and decouple file size from application memory consumption.

Administrative review workflow The ingestion and moderation workflow for externally sourced Projects requires a dedicated queue data model, an administrative API surface, and the ingestion logic that processes approved entries. This workflow introduces a new interaction pattern for asynchronous administrative decisions rather than real-time user operations that the job scheduler infrastructure would naturally support.

Security and compliance hardening The application does not yet implement request rate limiting at the API level. Computationally expensive endpoints, particularly format conversion, are prone to abuse. Adding rate limiting middleware or gateway-level throttling closes this gap. Separately, GDPR-compliant account management, such as structured user data export, cascading deletion of personal data, and cleanup of expired anonymous sessions, will be required before institutional deployment.

7.3 Potential Architectural Developments

A third group of ”next steps” would change the structure of the system itself. These are not immediate development priorities. They are paths that become appropriate when specific conditions are met.

Stronger structural invariant enforcement The current implementation realizes aggregate consistency primarily through application-service coordination rather than from behavior-rich domain objects. While this service-oriented approach keeps the implementation pragmatic and accessible, future evolution could move selected lifecycle and con-

sistency rules closer to the aggregate model itself. Encapsulating transition logic and invariant enforcement more explicitly within domain abstractions would reduce reliance on procedural coordination across service boundaries and strengthen the self-describing nature of the domain model.

Formalized lifecycle transitions The current lifecycle model represents visibility primarily through project-level visibility indicators and the existence of Share Links. While sufficient for the currently implemented states, the model does not explicitly encode valid transitions or lifecycle invariants. As additional states such as embargoed, scheduled, or retracted publication are introduced, a more explicit transition model may become necessary to ensure that lifecycle-dependent access rules remain consistent across application and persistence layers.

Graph-oriented analytical representation The current persistence model stores analytical content as opaque JSON snapshots. This preserves format independence and avoids coupling the persistence schema to the structural variability of CAQDAS formats, but it also limits the system to archival and re-export functionality. Analytical structures such as code hierarchies, coded segments, or memo relationships are not queryable at the database level because they remain embedded within serialized snapshots.

Supporting analytical queries across Projects would require a structured representation of these relationships. One possible approach would be decomposing the internal corpus representation into explicitly persisted entities and relations, either through a relational graph encoding within PostgreSQL or through a dedicated graph database. Both approaches would introduce significant architectural complexity. A relational model would reintroduce schema coupling to evolving format semantics, while a graph database would require an additional persistence backend together with a separate consistency and access-control model.

A further challenge concerns versioning. The current snapshot strategy creates immutable versions of Projects through single serialized writes. A decomposed graph representation would require either snapshotting entire graph states per version or introducing fine-grained delta-based version tracking for nodes and edges.

The architectural relevance of this trade-off depends on the future role of QDArchive. If the system remains focused on preservation and interoperability, the current snapshot model remains appropriate. If the focus shifts toward computational analysis or cross-project exploration of qualitative corpora, the snapshot model becomes a structural limitation and would likely need to be complemented by a derived graph-oriented representation.

Modular decomposition criteria The modular monolithic architecture provides clear extraction points should scaling or operational requirements change. The Convert module is the most natural first candidate. It interacts with the Share module through a single well-defined operation and owns no shared database state. Extraction would require replacing the synchronous service call with an enqueue operation and introducing a messaging layer (Newman, 2021). The job scheduler infrastructure described in the previous section would be the natural host for this. The Find module is fully read-only and stateless. It could be served by a dedicated PostgreSQL read replica without any code change. The Share module, as the central consistency boundary, should be the last to decompose. Doing so would require distributing the state transitions and access control logic that the

monolith currently provides directly, which is only justified if observed load exceeds the capacity of a vertically scaled instance.

Long-term format migration As QDA tools evolve and the REFI-QDA standard is updated, the internal corpus representation stored in `artifact_data` may diverge from what current library versions can re-export. A format migration strategy where stored snapshots are periodically re-validated against the current library version and re-serialized if necessary would address the long-term preservation risk identified in the related work and is consistent with the OAIS model's requirement that content remain interpretable over time (CCSDS, 2024).

8 Conclusion

QDA artifacts consist of more than the source material on which they are based. They capture coding schemes, annotations, analytical relationships, and interpretive structures that emerge throughout the research process. Yet in most archival environments, these artifacts are reduced to opaque project files whose internal structure remains inaccessible to the archive itself. Preserving qualitative research therefore requires more than storing files. It requires preserving the analytical context, lifecycle, and semantics embedded within them.

QDArchive was developed from this perspective. The resulting architecture demonstrates how QDA artifacts can be managed as structured research objects with explicit lifecycle states, ownership semantics, and publication controls. As a consequence, preservation, interoperability, and controlled access emerge from a shared domain model rather than being implemented as separate technical capabilities.

8.1 Summary of Contributions

The primary contribution of this thesis is a domain-oriented architectural approach for managing qualitative research artifacts as structured research objects rather than opaque project files. To achieve this, the proposed model explicitly focuses on capturing the architectural aspects that are related to artifact lifecycle, ownership concerns, publication state, and version history. This provides a foundation for archival management beyond simple file storage.

A second contribution is the design and implementation of QDArchive as a modular monolithic backend architecture developed alongside QDConvert, a dedicated conversion library for qualitative research artifacts. Together, both systems address the interoperability challenges posed by heterogeneous and vendor-specific CAQDAS formats. While QDConvert focuses on format interoperability, QDArchive provides a domain model for managing artifacts independently of the underlying file format.

A third contribution is the requirement-driven evaluation of the resulting architecture. Using quality attributes and scenario-based reasoning, the evaluation demonstrates the feasibility of the proposed approach, highlights architectural trade-offs, and identifies areas for future development.

8.2 Architectural Reflection

An important insight gained during the design and implementation process was that the overall complexity of the domain does not originate from file storage or format conversion alone. Instead, complexity emerges from the requirement to keep the system in a consistent state while simultaneously managing access rules, publication states, version history, and interoperability concerns. Treating artifacts as domain entities was therefore crucial because it allowed the various states and rules to be consolidated in a central location instead of treating them as separate properties of individual files. This made the overall complexity more manageable and consistent.

The modular monolithic architecture proved to be a pragmatic response to these requirements. As mentioned above, the domain demands strong consistency between lifecycle transitions, access control decisions, and publication states. Maintaining this consistency in a distributed architecture would introduce an additional level of complexity. This is not

justifiable given the current scale. Another positive aspect of using a modular monolithic approach is the introduction of bounded contexts and explicit architectural boundaries. These boundaries reduce coupling between functional areas and make it easier to replace or adapt individual parts of the system should operational requirements change.

The implementation also revealed several architectural limitations that result from deliberate design and infrastructure choices. One of the most significant challenges concerns the management of transactional consistency. Because the selected infrastructure abstraction does not expose direct transaction control to the application layer, certain multi-step operations must be executed as sequences of independent database actions. In addition, the system's modular structure and domain invariants are primarily maintained through development conventions rather than automated enforcement mechanisms. Bounded contexts exist as conceptual architectural constructs, but their boundaries are not currently enforced through fitness functions or import restrictions. Business rules reside largely in service-layer procedures instead of in domain objects themselves. While this keeps the implementation pragmatic and readable, correctness depends on the disciplined application of established patterns instead of structural guarantees provided by the architecture itself. These observations are best understood not as contradictions of the proposed architecture, but as insights into the practical trade-offs involved in implementing it. They identify potential directions for future refinement of the architecture.

The central outcome of this thesis therefore extends beyond the implementation of a single system. The work suggests that aligning lifecycle management, access control, interoperability requirements, and preservation concerns within a coherent architectural model can provide a viable foundation for qualitative research repositories. More broadly, it suggests that domain-driven architectural approaches can help preserve not only research data itself, but also the analytical structures and knowledge embedded within qualitative research projects.

References

Antes, A., Walsh, H., Strait, M., Hudson-Vitale, C., & DuBois, J. (2017). Examining Data Repository Guidelines for Qualitative Data Sharing. *Journal of Empirical Research on Human Research Ethics*, *13*, 155626461774412.

Baker, M. (2016). 1,500 Scientists Lift the Lid on Reproducibility. *Nature*, *533*(7604), 452–454.

Bass, L., Clements, P., & Kazman, R. (2021). *Software Architecture in Practice* (4th ed.). Addison-Wesley.

Bishop, L. (2009). Ethical Sharing and Reuse of Qualitative Data. *Aust J Soc Issues*, *44*.

Bishop, L., & Kuula-Luumi, A. (2017). Revisiting Qualitative Data Reuse: A Decade On. *Sage Open*, *7*(1), 2158244016685136.

Borgman, C. L. (2012). The Conundrum of Sharing Research Data. *Journal of the American Society for Information Science and Technology*, *63*(6), 1059–1078.

CCSDS. (2024). "Reference Model for an Open Archival Information System (OAIS)" (tech. rep. No. CCSDS 650.0-M-3). Consultative Committee for Space Data Systems.

CERN. (n.d.). *Zenodo*. Last accessed: April 9, 2026 https://zenodo.org

CESSDA. (2017). *CESSDA Data Management Expert Guide*. Last accessed: May 5, 2026 https://dmeg.cessda.eu

Chandra, Y., & Shang, L. (2019). Computer-assisted qualitative research: An overview. In *Qualitative research using r: A systematic approach* (pp. 21–31). Springer Nature Singapore.

Corti, L. (2007). Re-using archived qualitative data - Where, how, why? *Archival Science*, *7*, 37–54.

Corti, L., & Gregory, A. (2011). CAQDAS comparability. What about CAQDAS data exchange? *Forum Qualitative Sozialforschung / Forum: Qualitative Social Research*, *12*.

DANS. (2024). *DANS – Data Archiving and Networked Services*. Last accessed: May 5, 2026 https://dans.knaw.nl

DDI. (2024). *Data Documentation Initiative (DDI)*. Last accessed: May 9, 2026 https://ddialliance.org

Evans, E. (2004). *Domain-Driven Design : tackling complexity in the heart of software* (1st ed.). Addison-Wesley.

Evers, J. (2018). Current Issues in Qualitative Data Analysis Software (QDAS): A User and Developer Perspective. *The Qualitative Report*.

Evers, J., Caprioli, M., Nöst, S., & Wiedemann, G. (2020). What is the REFI-QDA Standard: Experimenting With the Transfer of Analyzed Research Projects Between QDA Software. *Forum Qualitative Sozialforschung*, *21*.

Faniel, I., & Jacobsen, T. (2013). Reusing Scientific Data: How Earthquake Engineering Researchers Assess the Reusability of Data. *Computer Supported Cooperative Work*.

Faniel, I., Kansa, E., Kansa, S., Barrera-Gomez, J., & Yakel, E. (2013). The Challenges of Digging Data: A Study of Context in Archaeological Data Reuse. *Proceedings of the ACM/IEEE Joint Conference on Digital Libraries*, 295–304.

Fielding, R. T. (2000). *Architectural Styles and the Design of Network-based Software Architectures* [Doctoral dissertation, University of California, Irvine].

Figshare. (n.d.). *Figshare*. Last accessed: May 5, 2026 https://figshare.com

Ford, N., Parsons, R., Kua, P., & Sadalage, P. (2022). *Building Evolutionary Architectures* (2nd ed.). O'Reilly Media.

Foster, E., & Deardorff, A. (2017). Open Science Framework (OSF). *Journal of the Medical Library Association*, *105*.

Fowler, M. (2002). *Patterns of Enterprise Application Architecture* (1st ed.). Pearson International.

Fowler, M. (2003). *Anemic Domain Model*. Last accessed: May 13, 2026 https://martinfowler.com/bliki/AnemicDomainModel.html

Fowler, M. (2004). Inversion of Control Containers and the Dependency Injection Pattern. *martinfowler.com*.

Fowler, M. (2015). *Monolith First*. Last accessed: April 20, 2025 https://martinfowler.com/bliki/MonolithFirst.html

Garcia-Molina, H., & Salem, K. (1987). Sagas. *ACM SIGMOD Record*, *16*(3), 249–259.

GDPR. (2016). *Regulation (EU) 2016/679 (General Data Protection Regulation)*. Last accessed: May 5, 2026 https://eur-lex.europa.eu/eli/reg/2016/679/oj/eng

Gray, J., & Reuter, A. (1992). *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann.

Harvard Dataverse. (n.d.). *Harvard Dataverse*. Last accessed: May 5, 2026 https://dataverse.harvard.edu

Hevner, A. R., March, S. T., Park, J., & Ram, S. (2004). Design Science in Information Systems Research. *MIS Quarterly*, *28*(1), 75–105.

Hocker, J., Bipat, T., McDonald, D. W., & Zachry, M. (2021). Evaluating QualiCO: An Ontology to Facilitate Qualitative Methods Sharing to Support Open Science. *Journal of Internet Services and Applications*, *12*(5).

ICPSR. (n.d.). *Inter-university Consortium for Political and Social Research*. Last accessed: May 5, 2026 https://www.icpsr.umich.edu

Inter-university Consortium for Political and Social Research (ICPSR). (n.d.). *Qualitative Data Sharing (QDS) Project*. Last accessed: May 20, 2026 https://www.icpsr.umich.edu/web/ICPSR/series/1780

Ioannidis, J. P. A. (2005). Why Most Published Research Findings Are False. *PLOS Medicine*, *2*(8), e124.

ISO/IEC. (2023). *ISO/IEC 25010:2023 Systems and Software Engineering — Systems and Software Quality Requirements and Evaluation (SQuaRE): Product Quality Model*. International Organization for Standardization. https://www.iso.org/standard/35733.html

Jones, M., Bradley, J., & Sakimura, N. (2015). *RFC 7519: JSON Web Token (JWT)*. Last accessed: May 5, 2026 https://datatracker.ietf.org/doc/html/rfc7519

Karcher, S., Kirilova, D., & Weber, N. (2016). Beyond the matrix: Repository services for qualitative data. *IFLA Journal*, *42*.

Kazman, R., Klein, M., & Clements, P. (2000). Atam: Method for architecture evaluation.

Khononov, V. (2021). *Learning Domain-Driven Design: Aligning Software Architecture and Business Strategy*. O'Reilly Media.

Kleppmann, M. (2017). *Designing Data-Intensive Applications*. O'Reilly Media.

Leach, P. J., Mealling, M., & Salz, R. (2005). *A Universally Unique IDentifier (UUID) URN Namespace*. Last accessed: May 13, 2026 https://www.rfc-editor.org/info/rfc9562/

Mason, J. (2002). *Qualitative Researching* (2nd ed.). SAGE.

Microsoft. (2024). *TypeScript TSConfig Reference: strict*. https://www.typescriptlang.org/tsconfig#strict

Newman, S. (2021). *Building microservices : designing fine-grained systems* (2nd ed.). O'Reilly Media, Incorporated.

Nielsen, Jakob. (1994). *10 usability heuristics for user interface design*. Last accessed: May 5, 2026 https://www.nngroup.com/articles/ten-usability-heuristics/

Node.js. (2026a). *Node.js Documentation: About Node.js*. https://nodejs.org/learn/asynchronous-work/event-loop-timers-and-nexttick

Node.js. (2026b). *The Node.js Event Loop*. Last accessed: May 10, 2026 https://nodejs.org/learn/asynchronous-work/event-loop-timers-and-nexttick

OECD. (2007). *OECD Principles and Guidelines for Access to Research Data from Public Funding*.

Open Science Collaboration. (2015). Estimating the reproducibility of psychological science. *Science*, *349*.

OWASP Foundation. (2024). *Session Management Cheat Sheet*. Last accessed: May 5, 2026 https://cheatsheetseries.owasp.org/cheatsheets/Session_Management_Cheat_Sheet.html

Parnas, D. L. (1972). On the Criteria To Be Used in Decomposing Systems into Modules. *Communications of the ACM*, *15*(12), 1053–1058.

PostgreSQL. (2026a). *Full Text Search Functions and Operators*. Last accessed: April 17, 2026 https://www.postgresql.org/docs/current/functions-textsearch.html

PostgreSQL. (2026b). *GIN Indexes*. Last accessed: April 17, 2026 https://www.postgresql.org/docs/current/gin.html

PostgreSQL. (2026c). *JSON Types*. Last accessed: May 10, 2026 https://www.postgresql.org/docs/current/datatype-json.html

PostgreSQL. (2026d). *PostgreSQL Documentation: Constraints*. Last accessed: April 15, 2026 https://www.postgresql.org/docs/current/ddl-constraints.html

PostgreSQL. (2026e). *PostgreSQL documentation: Create view*. Last accessed: May 10, 2026 https://www.postgresql.org/docs/current/sql-createview.html

PostgreSQL. (2026f). *PostgreSQL Documentation: Row Security Policies*. Last accessed: April 15, 2026 https://www.postgresql.org/docs/current/ddl-rowsecurity.html

PostgreSQL. (2026g). *PostgreSQL Documentation: Transaction Isolation*. Last accessed: April 15, 2026 https://www.postgresql.org/docs/current/transaction-iso.html

Richards, M., & Ford, N. (2020). *Fundamentals of software architecture: An engineering approach*. O'Reilly Media.

Saldaña, J. (2021). *The Coding Manual for Qualitative Researchers* (4th ed.). SAGE.

Saltzer, J. H., & Schroeder, M. D. (1975). The Protection of Information in Computer Systems. *Proceedings of the IEEE*, *63*(9), 1278–1308.

Supabase. (n.d.). *Architecture*. Last accessed: April 15, 2026 https://supabase.com/docs/guides/getting-started/architecture

Supabase. (2021). *Allow long-running transactions at the client side*. https://github.com/orgs/supabase/discussions/526

Syracuse University. (n.d.). *Qualitative Data Repository (QDR)*. Last accessed: May 5, 2026 https://qdr.syr.edu

Tenopir, C., Allard, S., Douglass, K., Aydinoglu, A., Wu, L., Read, E., Manoff, M., & Frame, M. (2011). Data Sharing by Scientists: Practices and Perceptions. *PloS one*, *6*, e21101.

UK Data Service. (n.d.). *UK Data Service*. Last accessed: May 5, 2026 https://ukdataservice.ac.uk/

UNESCO. (2021). *UNESCO Recommendation on Open Science*.

Vercel. (2026). *Next.js Documentation.* Last accessed: May 10, 2026 https://nextjs.org/docs

Vernon, V. (2013). *Implementing Domain-Driven Design.* Addison-Wesley.

Vernon, V. (2016). *Domain-Driven Design Distilled.* Addison-Wesley.

Whyte, A., & Tedds, J. (2011). "Making the Case for Research Data Management" (DCC Briefing Papers). Digital Curation Centre. Edinburgh.

Wiggins, A. (2017). *The Twelve-Factor App.* Last accessed: May 5, 2026 https://12factor.net

Wilkinson, M. D., Dumontier, M., Aalbersberg, I. J., Appleton, G., Axton, M., Baak, A., et al. (2016). The FAIR Guiding Principles for scientific data management and stewardship. *Scientific Data*, *3*, 160018.