MASARYK
UNIVERSITY

FACULTY OF INFORMATICS

Provenance Storage System Implementation in Java

Master's Thesis

MARCEL KROŠLÁK

Brno, Spring 2026

MASARYK UNIVERSITY

FACULTY OF INFORMATICS

Provenance Storage System Implementation in Java

Master's Thesis

MARCEL KROŠLÁK

Advisor: RNDr. Rudolf Wittner

Department of Computer Systems and Communications

Brno, Spring 2026

FACULTAS ARTIS INFORMATICÆ • UNIVERSITAS MASARYKIANA •

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

During the preparation of this thesis, I used the following AI tools:

- Grammarly – for grammar check,
- Claude – to improve my writing style,
- Github Copilot – for faster code writing.

I declare that I used these tools in accordance with the principles of academic integrity. I checked the content and took full responsibility for it.

Marcel Krošlák

Advisor: RNDr. Rudolf Wittner

Acknowledgements

I would like to express my sincere thanks to my supervisor, RNDr. Rudolf Wittner, for his guidance, constructive feedback, and the time he devoted to this thesis.

Abstract

Provenance information describes the history of data, including how it was created, transformed, and exchanged between organizations. The Provenance Storage System is a Python implementation of a system for storing and managing provenance documents according to the Common Provenance Model. This thesis rewrites the system into Java as two Spring Boot microservices. The Provenance Storage Service handles storage, retrieval, validation, and versioning of CPM-compliant documents in a Neo4j graph database. The Trusted Party Service manages X.509 certificates, verifies digital signatures, and issues non-repudiation tokens signed with ECDSA. The system supports CRUD operations for provenance documents, maintains non-repudiation evidence for stored documents, and enables authenticity verification at three levels of granularity: full provenance document, domain-specific part, and traversal information. The implementation follows a modular layered architecture designed for extensibility. It is validated through automated tests covering service logic, repository operations, and API contracts.

Keywords

API design, REST API, Java, Python, W3C PROV, Common Provenance Model

Contents

Introduction **1**

1 Background **3**
1.1 PROV Data Model 3
1.2 Common Provenance Model 5
1.2.1 Provenance Components and Backbone 5
1.2.2 Traversal Information Structures 7
1.2.3 Meta-Component and Versioning 8
1.3 Integrity and Non-repudiation 9
1.3.1 Digital Signatures and Certificate Chains 9
1.3.2 Non-repudiation Policy 10
1.4 Predecessor System 11
1.4.1 Architecture Overview 11
1.4.2 Core Functionality 12
1.4.3 Subsequent Refinements 12

2 Design and Analysis **13**
2.1 Functional Requirements 13
2.2 System Architecture 15
2.2.1 Operating Modes 16
2.3 Provenance Storage Service 18
2.3.1 API Design 18
2.3.2 Document Persistence 19
2.3.3 Document Validation 19
2.3.4 Meta-component 21
2.4 Trusted Party Service 21
2.4.1 API Design 21
2.4.2 Certificate Chain of Trust 22
2.4.3 Non-repudiation Token Structure 22
2.4.4 Token Issuance Process 23
2.4.5 Evidence Retention 24
2.5 Differences from Predecessor System 24

3 Implementation **26**
3.1 Technology Stack 26
3.2 Project Structure 27

3.3 Provenance Storage Service 28
3.3.1 REST Endpoints 28
3.3.2 Data Model 29
3.3.3 Document Processing 30
3.3.4 CPM Validation 30
3.3.5 Meta-component Management 31
3.3.6 Subgraph Generation 31
3.3.7 Trusted Party Communication 32
3.3.8 Document Store Flow 32
3.4 Trusted Party Service 34
3.4.1 REST Endpoints 34
3.4.2 Data Model 35
3.4.3 Certificate Verification 35
3.4.4 Token Construction and Signing 36
3.4.5 Signature Verification 37
3.4.6 Evidence Persistence 37
3.5 Error Handling 37
3.6 Configuration and Deployment 38
3.6.1 Provenance Storage Service 39
3.6.2 Trusted Party Service 39
3.6.3 Docker Compose Deployment 40
3.6.4 Monitoring 40
3.7 Limitations 41

4 Tests **42**
4.1 Test Frameworks and Infrastructure 42
4.1.1 Neo4j Test Harness 42
4.1.2 Test Data Factory 43
4.1.3 Test Profiles 43
4.2 Unit Tests 43
4.2.1 Provenance Storage Service 44
4.2.2 Trusted Party Service 44
4.2.3 Repository Slice Tests 45
4.2.4 Utility Tests 45
4.3 Component and Integration Tests 45
4.3.1 Provenance Storage Service 45
4.3.2 Trusted Party Service 46
4.4 Test Coverage Summary 46

4.5 Limitations of the Test Suite 47

5 Future Work **48**

6 Conclusion **50**

Bibliography **52**

A Electronic Archive Structure **54**

List of Tables

3.1 REST endpoints of the Provenance Storage Service. 28
3.2 REST endpoints of the Trusted Party Service. 34
3.3 Exception-to-HTTP-status mapping in the Provenance Storage Service. 38
4.1 Test suite structure and coverage areas. 47

List of Figures

1.1 PROV-DM core structures: Entity, Activity, and Agent connected by directed relations. Arrows are read in their direction, e.g., an Activity *used* an Entity. Picture taken from [2]. 4
1.2 A CPM provenance component. The upper part contains traversal information (main activity and forward connector). The lower part contains domain-specific provenance describing the actual processing. Picture taken from [3]. . 6
1.3 An example of a provenance chain. Multiple CPM provenance components from different organizations are linked via matching connector identifiers, forming a traversable chain. Picture taken from [3]. 6
1.4 CPM traversal information structure. The main activity uses backward connectors and generates forward connectors. Sender and receiver agents express inter-organizational responsibility. Picture taken from [3]. 8
2.1 High-level system architecture. The Provenance Storage Service handles document operations and communicates with the Trusted Party Service for signature verification and token issuance. Both services have independent databases. 15
2.2 Document storage flow with Trusted Party enabled. The service validates the document, verifies the signature via the Trusted Party, requests a token, and persists the graph with meta-component. 17
2.3 Document storage flow with Trusted Party disabled. The service validates the document and persists it without signature verification or token issuance. 17
2.4 Example of a non-repudiation of origin token issued by the Trusted Party Service (sensitive values redacted). . . 23

Introduction

Provenance describes the origin and processing history of data, including the entities, activities, and agents involved in its creation, transformation, and exchange. In information systems, provenance enables traceability of results, assessment of data quality, reproducibility of processing workflows, and accountability of involved parties [1]. The World Wide Web Consortium[1] (W3C) PROV standards provide a formal and interoperable framework for representing such information across heterogeneous systems [2].

In practice, data processing increasingly spans multiple organizations, each of which manages only a partial view of the overall processing history. This fragmentation creates a need for standardized mechanisms that allow provenance records from different organizations to be linked into coherent chains. The Common Provenance Model (CPM) addresses this need by extending PROV with traversal-oriented structures that support cross-organizational provenance chaining, enabling both backward traceability to verify a data source and, where available, forward traceability toward downstream processing. CPM also distinguishes between traversal-oriented provenance, which supports cross-component navigation, and domain-specific provenance, which captures processing details within a single organization [3].

In multi-organizational settings, accountability extends beyond syntactic validity of provenance records. An organization that publishes a provenance document may later have a motive to deny or alter its content, for example, if data it provided turns out to be incorrect and another organization has already relied on it [4]. Integrity and non-repudiation mechanisms are therefore necessary to ensure that provenance documents remain verifiable and that authorship cannot be plausibly denied [5].

A provenance storage system addressing these requirements was previously implemented in Python [4], and a subsequent thesis extended its testing coverage and refined selected implementation aspects [6]. However, wide adoption of the CPM requires implementations in multiple programming languages, including Java.

1. `https://www.w3.org/`

This thesis therefore presents a Java-based implementation of the distributed provenance storage system with Trusted Party integration, aligned with current CPM conventions. The work focuses on modular service architecture, standards-oriented provenance validation, and test-supported quality assurance.

The remainder of the thesis is organized as follows. Chapter 1 presents the technical background, including the PROV data model, CPM structures, and cryptographic foundations. Chapter 2 describes the system design and requirements analysis. Chapter 3 details the implementation and deployment. Chapter 4 summarizes the testing strategy and results, and Chapter 5 outlines potential extensions. Finally, Chapter 6 concludes the thesis.

1 Background

This chapter presents the technical foundations necessary for understanding the design and implementation of the provenance storage system. It introduces the W3C PROV data model, CPM and its structures for cross-organizational provenance chaining, the cryptographic basis for integrity and non-repudiation, and the architecture of the predecessor Python system on which this work builds.

1.1 PROV Data Model

The PROV Data Model (PROV-DM) is the central specification of the W3C PROV family and defines a conceptual model for representing provenance as directed graphs [2]. A provenance graph consists of three types of nodes:

- **Entity** – a data artifact, object, or resource whose provenance is being described.
- **Activity** – a process or transformation that creates, uses, or modifies entities.
- **Agent** – an actor (person, organization, or software) responsible for activities.

These nodes are connected by directed relations expressing causality and responsibility. The relations relevant to this thesis are:

- *wasGeneratedBy* – an entity was created by an activity.
- *used* – an activity consumed an entity.
- *wasDerivedFrom* – one entity was produced from another.
- *wasAssociatedWith* – an agent was responsible for an activity.
- *wasAttributedTo* – an entity is attributed to an agent.
- *wasInformedBy* – one activity communicated with another.

- *actedOnBehalfOf* – delegation of responsibility between agents.
- *wasRevisionOf* – an entity is a revised version of another.
- *wasInfluencedBy* – one element had an effect on another.

The core structures and their relations are shown in Figure 1.1.

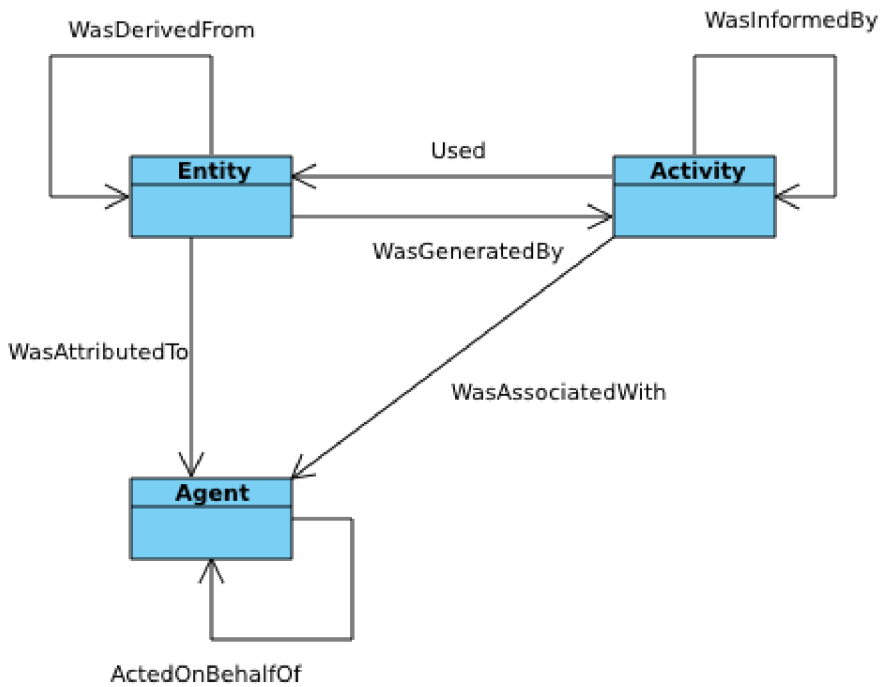

Figure 1.1: PROV-DM core structures: Entity, Activity, and Agent connected by directed relations. Arrows are read in their direction, e.g., an Activity *used* an Entity. Picture taken from [2].

PROV-DM also defines extended structures. A **Bundle** is a named set of provenance statements that can itself be treated as an entity, enabling meta-component (provenance about provenance).

Provenance graphs are exchanged using serialization formats defined by the PROV family: PROV-JSON and PROV-XML for structured machine processing, RDF via the PROV Ontology (PROV-O) for semantic web compatibility, and the PROV Notation (PROV-N) as a

human-readable format [7, 8]. Identifiers within provenance documents use qualified names composed of a namespace prefix and a local part, forming resolvable URIs.

1.2 Common Provenance Model

CPM extends PROV-DM with conventions for exchanging provenance across organizational boundaries [3]. In distributed workflows, each organization produces provenance only for the processing segment it performed. CPM defines how these individual provenance components, each implemented as a PROV bundle, are interconnected into a traversable provenance chain.

1.2.1 Provenance Components and Backbone

Each CPM provenance component (bundle) is divided into two parts:

- **Traversal information** – standardized structures defined by CPM that enable navigation between components across organizations.

- **Domain-specific provenance** – PROV-DM compliant content describing the actual processing performed within one organization. Its structure is not constrained by CPM beyond PROV-DM validity.

The traversal information portions of connected components together form the **provenance backbone**, an abstract chain that can be navigated without accessing domain-specific details. An example of a provenance component showing both parts is illustrated in Figure 1.2. An example of a provenance chain composed of multiple linked components can be seen in Figure 1.3.

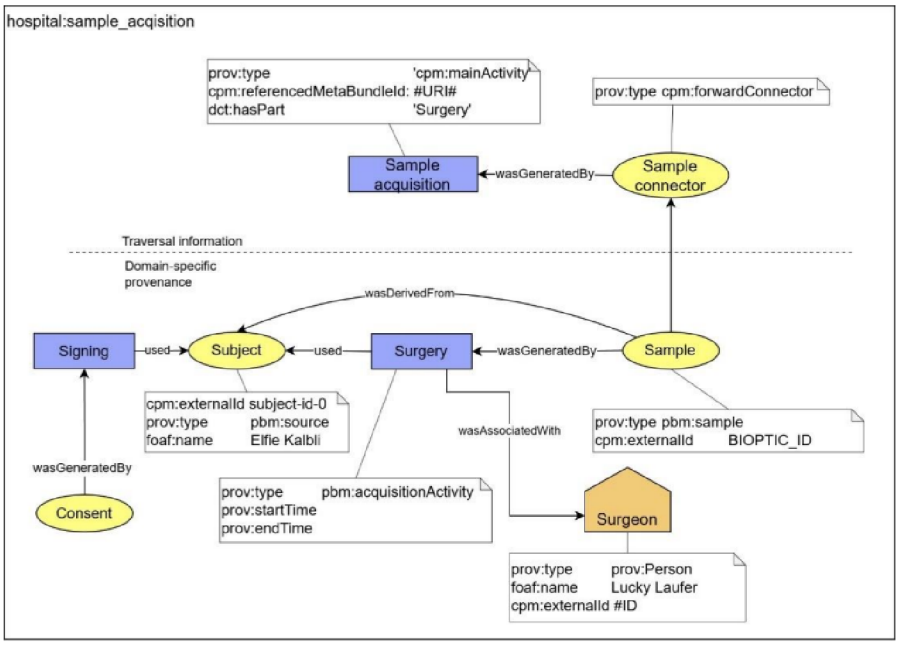

Figure 1.2: A CPM provenance component. The upper part contains traversal information (main activity and forward connector). The lower part contains domain-specific provenance describing the actual processing. Picture taken from [3].

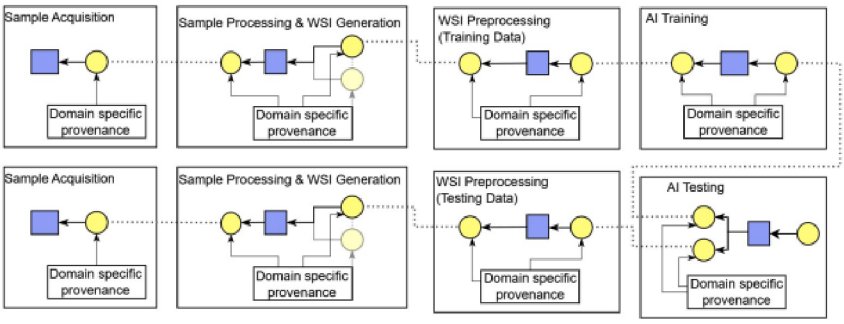

Figure 1.3: An example of a provenance chain. Multiple CPM provenance components from different organizations are linked via matching connector identifiers, forming a traversable chain. Picture taken from [3].

1.2.2 Traversal Information Structures

The traversal information within a component consists of the following elements, shown in Figure 1.4:

- **Main Activity** – represents the activity documented in the component. It carries a `cpm:referencedMetaBundleId` attribute pointing to the component's meta-component.
- **Backward Connector** – an entity representing input received from a preceding component. It carries attributes identifying the referenced bundle (`cpm:referencedBundleId`), its meta-component (`cpm:referencedMetaBundleId`), and a hash of the referenced document (`cpm:referencedBundleHashValue`).
- **Forward Connector** – an entity representing output produced by the main activity, serving as a connection point for subsequent components. CPM distinguishes a general forward connector (generated by the main activity) and specialized forward connectors attributed to specific receiver agents.
- **Forward Connector Specialized** – a forward connector representing a specific described object that has been sent from a sender to a receiver. It specializes the general forward connector.
- **Sender Agent** – in the receiver's traversal information, represents the organization that sent the data.
- **Receiver Agent** – in the sender's traversal information, represents the organization receiving the data.
- **Current Agent** – an agent representing the provenance controller of the current component.

Components are linked by matching connector identifiers. A backward connector in one component shares its identifier with the corresponding forward connector in the preceding component. This enables backward traceability (navigating to the source of received data) and, where forward connectors reference receiver bundles, forward traceability (navigating to downstream consumers).

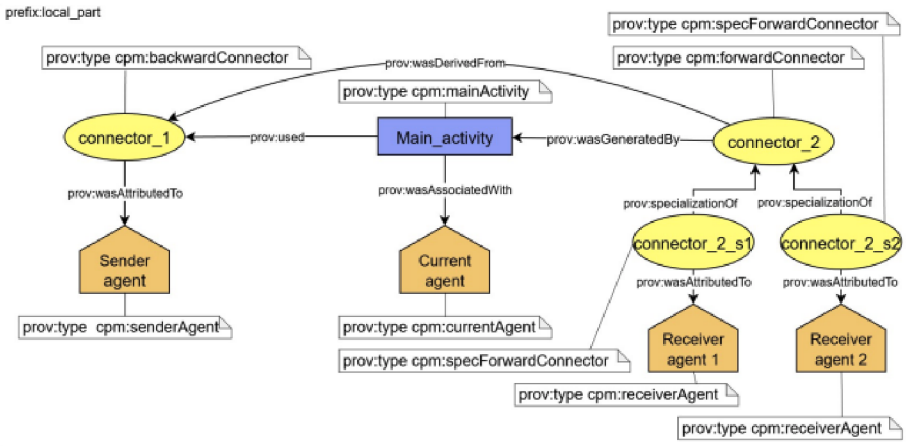

Figure 1.4: CPM traversal information structure. The main activity uses backward connectors and generates forward connectors. Sender and receiver agents express inter-organizational responsibility. Picture taken from [3].

1.2.3 Meta-Component and Versioning

Once a provenance component is finalized, it is considered immutable, which allows cryptographic operations such as hashing and signing to be applied. However, immutability means errors cannot be corrected in place. CPM addresses this through versioning. A new version of a component replaces the previous one, and this version history is recorded in a **meta-component** [3].

Meta-component uses PROV's bundle-as-entity mechanism. Each version of a component is represented as a specialized entity linked by *prov:revisionOf* relations, forming a version chain. The older versions are kept archived because other components may still reference them through their connectors.

A meta-component contains the following structures:

- A generator entity representing the document as an abstract concept, independent of any specific version.
- A version entity representing the stored version, linked to the generator entity via *specializationOf* and marked with a version number. Subsequent versions are linked to the previous version via *wasRevisionOf*.

- Optionally, a token entity recording non-repudiation token data, linked to a token generation activity associated with a Trusted Party agent.

1.3 Integrity and Non-repudiation

In a system where multiple organizations exchange provenance documents, two cryptographic properties are required beyond syntactic validity. **Integrity** ensures a document has not been altered since creation. **Non-repudiation of origin** prevents an author from denying that they produced the document [5].

To support non-repudiation, the system involves a **Trusted Party**, an independent, neutral entity distinct from both the provenance storage provider and the provenance generating organization. Each provenance storage instance is associated with one Trusted Party, which is trusted by the organizations that submit provenance documents to that instance.

1.3.1 Digital Signatures and Certificate Chains

Integrity can be achieved by computing a cryptographic hash of the finalized provenance document. Non-repudiation is supported through digital signatures. The provenance generating organization signs the document with its private key, and any party can verify the signature using the organization's public certificate [5].

Certificates are organized in a chain of trust. Each organization holds a certificate signed by a certificate authority (CA). Verification involves checking the entire chain from the organization's certificate up to a trusted root CA. This allows the system to confirm both the identity of the signer and the integrity of the signed content.

The Trusted Party issues a **non-repudiation token**, a signed artifact that binds a provenance document to its originator and creation time. **Evidence** is the broader set of information needed for dispute resolution, comprising the token, the original document, the submission context, and the certificates of the involved parties.

1.3.2 Non-repudiation Policy

Non-repudiation cannot be achieved through technical means alone. It also requires a clearly defined policy that governs the operational context [9]. The policy used in this system, established in [4] and based on [9], specifies that:

1. The Trusted Party is the only entity authorized to issue valid evidence. All evidence must be digitally signed by the Trusted Party, ensuring its authenticity and integrity.

2. There is always a possibility of private key compromise. The provenance generating organization must define a clearance period for reporting a confidentiality violation of its signing key. After the clearance period expires, all prior digital signatures are considered valid.

3. Tokens must include a timestamp generated by the Trusted Party.

4. The provenance storage provider cannot be the only party responsible for storing evidence, as it may have a motive to destroy it.

5. Evidence must be held in a way that prevents modification of stored content.

6. All information needed for integrity and authenticity verification must be sealed within the evidence.

7. A user that retrieves a provenance document can verify its meaningfulness, integrity, and authenticity, and should store it for prospective future claims.

8. The generated evidence must contain an identifier of the Trusted Party together with a unique identifier of the provenance storage provider.

9. The evidence generated before the organization stores the provenance graph must contain information about the time the document was created.

1.4 Predecessor System

The Java implementation presented in this thesis builds on a predecessor system developed in Python. This section describes the predecessor's architecture and functionality to provide the baseline against which the current system is understood.

1.4.1 Architecture Overview

The predecessor system consists of two independent REST API web services:

- **Provenance Storage Service** – the service with which client organizations interact. It handles storage, retrieval, and updating of provenance documents, generates meta-component, and communicates with the Trusted Party for non-repudiation token issuance. It uses Neo4j as its database for provenance document storage.

- **Trusted Party Service** – provides non-repudiation of origin by verifying document signatures, issuing non-repudiation tokens, and managing organization certificates. It uses PostgreSQL for storing certificates and token metadata.

Both services were implemented using Django[1] (Python web framework) and communicate over HTTPS. The provenance storage service uses the `prov`[2] Python library for parsing and serializing PROV documents and the `neomodel`[3] library for Neo4j interaction. Cryptographic operations (signature verification, certificate chain validation) rely on OpenSSL[4] bindings.

The system is stateless and designed for multi-instance deployment. Multiple provenance storage instances can operate at different physical locations, each paired with a Trusted Party of its choosing.

1. `https://www.djangoproject.com/`
2. `https://pypi.org/project/prov/`
3. `https://neomodel.readthedocs.io/`
4. `https://www.openssl.org/`

1.4.2 Core Functionality

The predecessor system provides the following operations:

- **Organization registration** – organizations register by providing their X.509 certificate, intermediate certificates, and a clearance period. They may specify a Trusted Party they trust.
- **Document storage** – a registered organization submits a base64-encoded provenance document together with its digital signature, a creation timestamp, and the document format. The service validates the document against constraints (correct bundle identifier format, presence of required CPM traversal elements, normalization), verifies the signature via the Trusted Party, and stores the graph in Neo4j. A non-repudiation token is returned upon success.
- **Document retrieval** – documents can be retrieved in full, or as separate traversal information and domain-specific parts. Each response includes the corresponding non-repudiation token.
- **Document update** – a newer version of a document can be stored, triggering meta-component and connector table updates.
- **Meta-component generation** – version chains are maintained as meta-components, recording the history of each component.
- **Connector table generation** – tables mapping forward and backward connectors between components are generated to support provenance chain traversal.

1.4.3 Subsequent Refinements

The predecessor system has been extended [6] by adding PROV validity checking (verifying that stored documents are valid PROV graphs), CPM-specific structural validation (verifying presence and correctness of traversal information elements), and comprehensive unit and component test coverage. These refinements also updated the system to align with changes in the CPM specification that occurred after the original implementation.

2 Design and Analysis

This chapter presents the design of the Java-based provenance storage system. It first specifies the functional requirements, then describes the system architecture and the design of both services, including their APIs, validation logic, document lifecycle, and operating modes. At the end, the chapter summarizes the key differences from the predecessor system described in Section 1.4.

2.1 Functional Requirements

The functional requirements are derived from the predecessor system [4] and refined based on updated CPM conventions [3] and changes introduced in [6]. In addition, the design must satisfy the non-repudiation policy defined in Section 1.3.2. The following list defines the externally observable capabilities of the system.

- **Organization registration.** Organizations must register before storing provenance documents. During registration, the organization provides its X.509 certificate and intermediate certificates, establishing the identity binding required for non-repudiation.
- **Certificate lifecycle management.** The system must support renewal and replacement of organization certificates, covering both routine expiration and emergency replacement in case of private key compromise.
- **Provenance document storage.** The system accepts provenance documents submitted by registered organizations. When Trusted Party integration is enabled, the request must include a digital signature and a creation timestamp. The system validates the document, verifies the signature, and issues a non-repudiation token. The original document must be preserved without modification because it is signed.
- **Provenance document update.** The system supports storing a new version of a previously stored document. The version history is tracked in meta-component

- **Provenance document retrieval.** Previously stored documents can be retrieved by identifier. Retrieval responses include the associated non-repudiation token when available.

- **Subgraph retrieval.** Beyond full-document retrieval, the system provides separate access to the traversal information and domain-specific parts of a CPM component. This is required for provenance chain navigation, where a consuming organization may need only the traversal information without accessing domain-specific content.

- **Document validation.** Before storage, documents are validated for PROV-DM compliance and CPM structural correctness. Documents that fail validation are rejected. Documents are required to be valid but not necessarily pre-normalized, since normalization could alter the signed content.

- **CPM structural compliance.** Submitted provenance components must contain a correctly formed main activity, backward connectors with all mandatory CPM attributes, and forward connectors with valid structure. Relations between traversal information elements must follow CPM conventions.

- **Non-repudiation of origin.** The system must support issuance of non-repudiation tokens signed by the Trusted Party, binding the document content, originator identity, and creation timestamp as evidence.

- **Integrity verification.** Organizations retrieving documents must be able to verify that the content has not been modified since storage, using the document digest recorded in the non-repudiation token.

- **Meta-component support.** The system must generate and maintain meta-components that track the version history of provenance documents, as required by CPM.

- **Evidence retention.** The system must retain evidence necessary for dispute resolution, including the provenance document, non-repudiation token, and organization certificates.

- **Multi-instance deployment.** The system must support deployment as multiple independent instances, where each instance holds a different segment of the provenance chain and may use a different Trusted Party.

2.2 System Architecture

The system consists of two independent REST API services, preserving the two-service decomposition established in the predecessor system. The high-level architecture is shown in Figure 2.1.

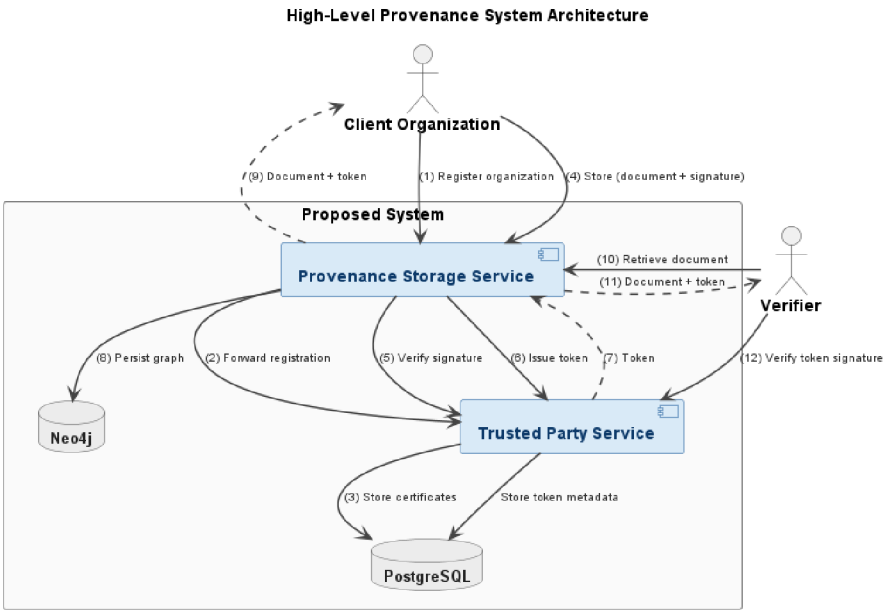

Figure 2.1: High-level system architecture. The Provenance Storage Service handles document operations and communicates with the Trusted Party Service for signature verification and token issuance. Both services have independent databases.

- **Provenance Storage Service** is the service with which client organizations interact. It is responsible for document storage, retrieval, validation, meta-component management, and subgraph generation. It persists provenance graphs in a graph database.

- **Trusted Party Service** provides the non-repudiation layer. It manages organization certificates, verifies document signatures, issues non-repudiation tokens, and retains evidence. It uses a relational database for persistence.

Both services are stateless and communicate over a REST API. The Provenance Storage Service calls the Trusted Party Service during store and update operations to verify signatures and obtain tokens. Each service can be deployed in multiple instances independently.

2.2.1 Operating Modes

The system supports two operating modes with respect to Trusted Party integration.

- **Trusted Party enabled** (default). Organization registration is required. Store and update requests must include a signature and creation timestamp. The Provenance Storage Service calls the Trusted Party to verify the signature and issue a token. Retrieval responses include the token.

- **Trusted Party disabled.** The system stores and retrieves documents without signature verification or token issuance. Organization registration is not available. This mode is primarily intended for development and testing scenarios.

The document storage flow for both modes is illustrated in Figures 2.2 and 2.3.

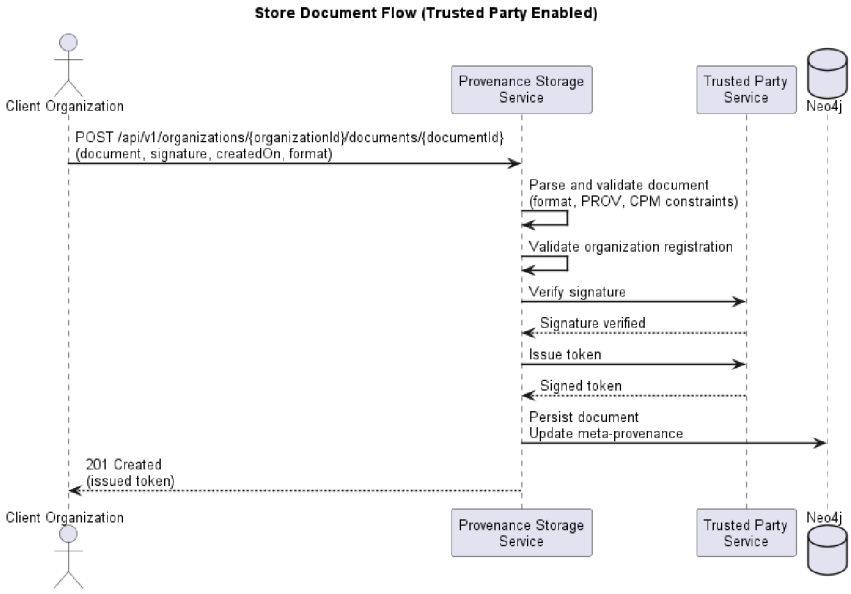

Figure 2.2: Document storage flow with Trusted Party enabled. The service validates the document, verifies the signature via the Trusted Party, requests a token, and persists the graph with meta-component.

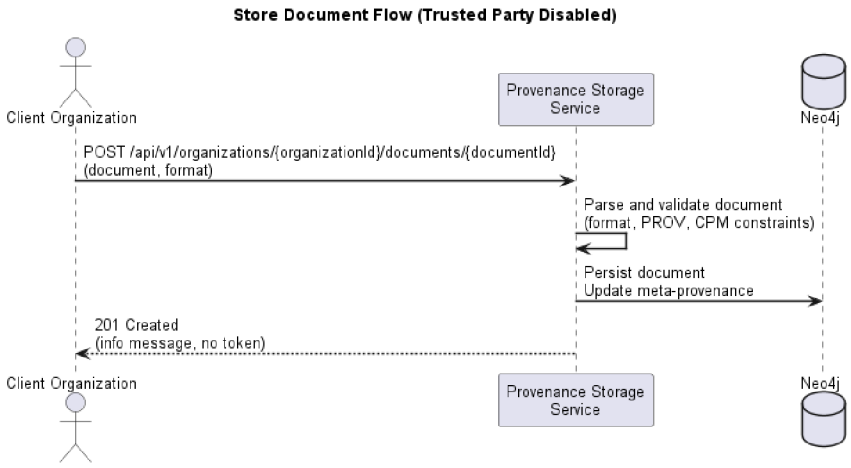

Figure 2.3: Document storage flow with Trusted Party disabled. The service validates the document and persists it without signature verification or token issuance.

2.3 Provenance Storage Service

The Provenance Storage Service exposes a REST API organized around three resource groups: organizations, documents, and meta-components. Documents are scoped to organizations, reflecting that each organization manages its own provenance components.

2.3.1 API Design

The API provides the following categories of operations:

- **Organization management.** Registering and updating organizations, including their certificate data and Trusted Party URI. Available only when Trusted Party is enabled.

- **Document operations.** Storing a new provenance document, updating an existing one with a new version, retrieving a stored document, and checking whether a document exists. Store and update requests carry the document content, its serialization format, and when Trusted Party is enabled, a signature and creation timestamp. Retrieval includes the associated non-repudiation token when available.

- **Subgraph retrieval.** Returning the traversal information or domain-specific part of a stored component separately. This supports provenance chain navigation where a consuming organization needs only the traversal information.

- **Meta-component operations.** Retrieving and checking existence of meta-components, which record the version history of documents.

When Trusted Party is enabled, store and update responses include the issued non-repudiation token. Retrieval responses also include the token. When Trusted Party is disabled, token fields are absent from responses.

2.3.2 Document Persistence

Provenance documents are persisted in their original serialized form. A `Document` node in Neo4j stores the full document as a string property alongside a format identifier. This matches the approach of the predecessor Python implementation, where the entire graph was stored as a node attribute.

The meta-component is handled differently: each version entity, token entity, activity, and agent is stored as a distinct Neo4j node with PROV relationships between them. This allows the version history to be queried directly as a graph without deserializing document content.

Subgraphs (traversal information and domain-specific parts) are computed from the stored serialized document on demand and cached as separate `Document` nodes.

2.3.3 Document Validation

Before a document is persisted, the service performs a multi-stage validation.

1. **Format parsing.** The submitted document is parsed according to the declared format (PROV-JSON, PROV-XML, RDF, or PROV-N). If parsing fails, the request is rejected.

2. **Bundle extraction.** The service extracts the single bundle contained in the document. A provenance component is expected to contain exactly one bundle.

3. **Identifier consistency.** The bundle identifier extracted from the graph must match the document identifier in the request URI. This ensures that the API path and the document content refer to the same resource.

4. **CPM structural validation.** The service checks that the bundle contains a correctly formed traversal information structure. Specifically:

    - A main activity with type `cpm:mainActivity` must be present. It must include a `cpm:referencedMetaBundleId` attribute.

- Backward connectors must have all mandatory attributes: `cpm:referencedBundleId`, `cpm:referencedMetaBundleId`, `cpm:referencedBundleSpecV`, `cpm:referencedmetaBundleSpecV`, `cpm:referencedBundleHashValue`, and `cpm:hashAlg`.
- Forward connectors require only `prov:type` with value `cpm:forwardConnector`.
- Forward connectors specialized must have all mandatory attributes: `cpm:referencedBundleId`, `cpm:referencedMetaBundleId`, `cpm:referencedBundleSpecV`, `cpm:referencedmetaBundleSpecV`, `cpm:provenanceServiceUri`, `cpm:referencedBundleHashValue`, and `cpm:hashAlg`. Each must also specialize exactly one general forward connector via `prov:specializationOf`.
- The bundle must contain at least one agent with type `cpm:currentAgent`.
- The main activity must use only backward connectors and generate only forward connectors.
- Connectors must not reference the current bundle or its own meta-component.

5. **PROV validity.** The document is validated against PROV-DM constraints to verify that it represents a valid provenance graph [10].

For update operations, additional checks apply. The target document must already exist, the referenced meta-component must exist, and the meta-component reference must be consistent with the already established version chain. This prevents an update from linking a document to an unrelated meta-component bundle.

2.3.4 Meta-component

For each stored document, the service maintains a meta-component that records the version history of that document following the structure described in Section 1.2.3.

When a document is updated, the service adds a new version entity with an incremented version number, linked to the previous version via *wasRevisionOf*. Both versions remain linked to the same generator entity. Token-related provenance is added for the new version in the same manner as for the initial store.

Meta-components are retrievable through a dedicated endpoint, separate from document retrieval. This allows clients to access the version history without retrieving the full document content.

2.4 Trusted Party Service

The Trusted Party Service provides the cryptographic evidence layer of the system. It operates as an independent service that cooperates with Provenance Storage Service instances.

2.4.1 API Design

The API is organized into four functional areas:

- **Service information.** Exposes the Trusted Party identifier and public certificate. Used by peer services when constructing or verifying non-repudiation evidence.
- **Organization and certificate management.** Supports registering organizations, querying their data, and managing their certificates. Registration includes certificate chain validation.
- **Signature verification.** Verifies a document signature against the submitting organization's active certificate. Returns success or failure.
- **Token issuance and retrieval.** Issues a signed non-repudiation token for a submitted document. Previously issued tokens can be retrieved for evidence purposes.

2.4.2 Certificate Chain of Trust

Before an organization can participate in non-repudiation workflows, its certificate chain must be validated. During registration, the Trusted Party verifies the submitted client certificate and intermediate certificates against a configured set of trusted root certificates. Only if the chain validates successfully is the organization accepted.

When certificates are updated, previously active certificates are revoked and replaced by the newly validated ones. The certificate history is retained for later verification and dispute analysis.

2.4.3 Non-repudiation Token Structure

Each non-repudiation token consists of a data object and a digital signature over that data. The data fields are:

- **originatorId** – identifier of the submitting organization.
- **authorityId** – identifier of the Trusted Party that issued the token.
- **tokenTimestamp** – time of token issuance.
- **documentCreationTimestamp** – creation time provided by the submitting organization.
- **documentDigest** – SHA-256 hash of the submitted document.
- **additionalData** – bundle identifier, hash algorithm identifier, Trusted Party URI, and Trusted Party certificate.

The token signature is computed over a canonicalized (deterministically sorted) JSON representation of the token data, using ECDSA[1] with SHA-256. This makes token verification deterministic across systems. The Trusted Party certificate needed for signature verification is included in the token metadata and is also available through the service information endpoint.

An example of an issued token is shown in Figure 2.4.

1. Elliptic Curve Digital Signature Algorithm

```
{
  "data": {
    "originatorId": "org1",
    "authorityId": "TrustedParty",
    "tokenTimestamp": "2026-04-03T13:13:15.6009072",
    "documentCreationTimestamp": "2024-04-03T12:00:00",
    "documentDigest": "5db5a69bf4cad3569bda405909b92b459627afb178a3bfe9bfa084c048ba6983",
    "additionalData": {
      "bundle": "http://example.org/doc1",
      "hashFunction": "SHA256",
      "trustedPartyUri": "127.0.0.1:8020",
      "trustedPartyCertificate": "-----BEGIN CERTIFICATE-----\r\n -CERTIFICATE- -----END CERTIFICATE-----\r\n"
    }
  },
  "signature": "MEUCIDNgcDyg3PeajVBR3d2EexMRcjiOb5gubHrwAbO4s96RAiEA7XPdR/MuQzHThmD4T5IEwWQnEgtVQEYqXaN9lIOPnps="
}
```


Figure 2.4: Example of a non-repudiation of origin token issued by the Trusted Party Service (sensitive values redacted).

2.4.4 Token Issuance Process

Token issuance follows a staged process:

1. The service validates the request fields (organization identifier, document, format, timestamp).
2. For document-type requests, the service verifies the submitted signature against the organization's active certificate.
3. The service parses the document and resolves the bundle identifier.
4. A SHA-256 digest of the document bytes is computed.
5. Token data is constructed containing originator, authority, temporal, and verification-context fields.
6. The token data is canonicalized to JSON and signed with the Trusted Party's private key using ECDSA.
7. The signed token is returned and the issuance record is persisted as evidence.

For meta-component tokenization, the same token structure is used but signature verification is not required as a precondition, since meta-component is generated by the service itself rather than submitted by an organization.

2.4.5 Evidence Retention

The Trusted Party retains evidence required for dispute resolution. This includes organization certificate history, issued token records, and submission context from token issuance. This retention satisfies the non-repudiation policy requirement that evidence must not be stored solely by the provenance storage provider.

2.5 Differences from Predecessor System

The Java system preserves the two-service architecture and core functionality of the predecessor but introduces the following design changes:

- **ISO-8601 timestamp support.** The predecessor API accepted document creation timestamps only as integer epoch seconds. The current system accepts both epoch seconds and ISO-8601 formatted strings.
- **Validation-first ordering.** The predecessor verified the document signature at the Trusted Party before parsing and validating the graph. The current system performs all local validation (parsing, CPM checks, PROV validity) first and only contacts the Trusted Party if the document passes. This avoids unnecessary network calls for invalid inputs.
- **PROV validity always enforced.** The predecessor had PROV standard validation disabled by default and relied on an external validation service when enabled. The current system always validates documents against PROV-DM constraints in-process before accepting them. The validation is performed by ProvToolbox's `Validate` class, which implements the W3C PROV-CONSTRAINTS specification [10]. The resulting `ValidationReport` is checked for malformed statements, type overlaps, cycles, qualified name mismatches, and failed merges. A document is rejected if any of these categories is non-empty.
- **No namespace validation.** The predecessor validated that all element identifiers use properly declared namespaces. The current system does not perform this check, relying on the docu-

ment parser to reject malformed identifiers during the parsing stage.

3 Implementation

This chapter describes how the design from Chapter 2 is realized in code. It covers the technology stack, project structure, and the implementation of both services. Concrete class names and short code examples are included where they clarify non-obvious decisions. The full source code is available on GitHub[1].

3.1 Technology Stack

The system is implemented in Java 21 using Spring Boot[2] 3.x. Both services share the same framework, dependency management, and build tool (Maven multi-module project). The following technologies are used:

- **Neo4j** (graph database) for provenance graph persistence in the Provenance Storage Service. Provenance naturally forms a directed graph of entities, activities, and agents connected by relations, which maps directly to a labeled property graph. Access is provided through Spring Data Neo4j[3].
- **PostgreSQL** (relational database) for the Trusted Party Service. Its data is record-oriented (organizations, certificates, tokens), making a relational model appropriate. Access is provided through Spring Data JPA[4] with Hibernate.
- **ProvToolbox**[5] for parsing, serializing, and validating W3C PROV documents across multiple formats (PROV-JSON, PROV-XML, RDF/Trig, PROV-N). Used in both services.
- **Bouncy Castle**[6] as the cryptographic provider for ECDSA signature operations and PEM key loading in the Trusted Party

1. `https://github.com/xkroslak/provenance-storage-java`
2. `https://spring.io/projects/spring-boot`
3. `https://spring.io/projects/spring-data-neo4j`
4. `https://spring.io/projects/spring-data-jpa`
5. `https://lucmoreau.github.io/ProvToolbox/`
6. `https://www.bouncycastle.org`

Service. Standard Java `CertPathValidator` is used for PKIX certificate chain verification.

- **Docker and Docker Compose** for containerized multi-service deployment including both application services, both databases, and a Prometheus/Grafana monitoring stack.

3.2 Project Structure

The repository is organized as a Maven multi-module project with two service modules sharing a parent POM. Each service follows a layered package structure:

```
provenance-storage-java/
  distributed-prov-system-service/
    src/main/java/cz/muni/fi/distributed_prov_system/
      rest/            -- REST controllers
      facade/          -- Facade interfaces and implementations
      service/         -- Business logic
      data/model/      -- Neo4j node and relationship entities
      data/repository/ -- Spring Data Neo4j repositories
      client/          -- REST client for Trusted Party calls
      utils/prov/      -- Validation and PROV processing
      api/             -- Request/response DTOs
      config/          -- AppProperties, bean configuration
  trusted-party-service/
    src/main/java/cz/muni/fi/trusted_party/
      rest/            -- REST controllers
      facade/          -- Facade interfaces and implementations
      service/         -- Business logic
      data/model/      -- JPA entities
      data/repository/ -- Spring Data JPA repositories
      mappers/         -- DTO-to-entity mappers
      utils/           -- Crypto and PROV utilities
      api/             -- Request/response DTOs
      config/          -- AppProperties
```

Both services follow the same layered architecture: controllers receive HTTP requests, facades provide thin orchestration boundaries,

services implement core processing logic, and repositories encapsulate database access. Controllers are organized by resource (documents, organizations, meta-component in the Provenance Storage Service and organizations, tokens, info in the Trusted Party Service).

3.3 Provenance Storage Service

3.3.1 REST Endpoints

The Provenance Storage Service exposes its API through three controller classes. Table 3.1 lists all endpoints with their HTTP methods. All paths share the common prefix `/api/v1/`.

Table 3.1: REST endpoints of the Provenance Storage Service.

URI pattern	Method	Purpose
`/organizations/{orgId}`	POST	Register organization
`/organizations/{orgId}`	PUT	Update organization
`/organizations/{orgId}/documents/{docId}`	POST	Store new document
`/organizations/{orgId}/documents/{docId}`	PUT	Update existing document
`/organizations/{orgId}/documents/{docId}`	GET	Retrieve document
`/organizations/{orgId}/documents/{docId}`	HEAD	Check document existence
`/organizations/{orgId}/documents/{docId}/backbone`	GET	Retrieve traversal information
`/organizations/{orgId}/documents/{docId}/domain-specific`	GET	Retrieve domain-specific part
`/documents/meta/{metaId}`	GET	Retrieve meta-component
`/documents/meta/{metaId}`	HEAD	Check meta-component existence

Store and update requests accept a JSON body containing the base64-encoded document, its format (`PROVN`, `JSON`, `XML`, or `TRIG`), and when Trusted Party is enabled, a `signature` and `createdOn` timestamp.

Subgraph and meta-component retrieval endpoints accept a `format` query parameter to select the output serialization.

3.3.2 Data Model

The PROV data model is mapped to Neo4j using Spring Data Neo4j annotations. All PROV node types extend a common base class `BaseProv`, which defines the shared structure:

Listing 3.1: Base class for all PROV nodes in Neo4j.

```
@Node
public class BaseProv {
    @Id
    private String identifier;

    @CompositeProperty(prefix = "attr_")
    private Map<String, Object> attributes;
}
```

The `identifier` field serves as the node's primary key and corresponds to the PROV qualified name. The `attributes` map stores all PROV attributes (types, CPM-specific fields) as a composite property with a prefix to avoid Neo4j property name collisions.

Concrete PROV types extend `BaseProv`:

- `Entity`, `Activity`, `Agent`, `Bundle` for PROV core structures.
- `ForwardConnector` and `BackwardConnector` extend `Entity` to allow typed repository queries on CPM connectors.

PROV relations are modeled using Spring Data Neo4j `@RelationshipProperties`. Each relation class (e.g., `WasGeneratedBy`, `WasDerivedFrom`, `Used`) holds an optional identifier, an attributes map, and a `@TargetNode` reference to the target PROV node. Relations are stored as typed edges in the Neo4j graph.

In addition to PROV graph data, the service stores operational entities in the same Neo4j database:

- `Document` – stores the original base64-encoded document content and format.

- `Token` – stores non-repudiation token fields (signature, digest, timestamps, additional data).
- `Organization` and `TrustedParty` – store registration and certificate data.

3.3.3 Document Processing

Document parsing and initial processing is handled by the `InputGraphChecker` class. This class encapsulates the full pre-persistence pipeline described in Section 2.3.3:

1. Parses the base64-encoded document using ProvToolbox (`ProvToolboxUtils.parseDocument`).
2. Extracts the single bundle from the parsed document.
3. Identifies the main activity, forward connectors, and backward connectors by their CPM type attributes.
4. Resolves the meta-component identifier from the main activity.

After parsing, `InputGraphChecker` exposes the extracted components for subsequent validation and persistence steps.

Format conversion between PROV serializations is handled by `ProvToolboxUtils`, which wraps the ProvToolbox `InteropFramework`. A custom Jackson deserializer (`CustomAttributeMapDeserializer`) overrides the default ProvToolbox JSON parsing behavior to preserve qualified name semantics in attribute maps during deserialization.

3.3.4 CPM Validation

CPM structural validation is implemented in `CPMValidatorImpl`. It performs the checks described in Section 2.3.3:

- Verifies that all backward connectors have the six mandatory attributes (`cpm:referencedBundleId`, `cpm:referencedMetaBundleId`, `cpm:referencedBundleSpecV`, `cpm:referencedmetaBundleSpecV`, `cpm:referencedBundleHashValue`, `cpm:hashAlg`).

- Verifies that forward connectors require only `prov:type` with value `cpm:forwardConnector`.
- Checks generation, usage, and derivation relations against the main activity to enforce CPM traversal constraints.
- Verifies that connectors do not reference their own bundle or meta-component.

PROV standard validity is checked by `ProvDocumentValidatorImpl`, which uses ProvToolbox's built-in validation to detect malformed statements, type overlaps, and constraint violations.

3.3.5 Meta-component Management

Meta-component creation and update is handled by `ImportGraphService`. This service is transactional and executes the version-chain logic described in the Design chapter:

For initial storage, it creates a generator entity (representing the document as an abstract concept), a first version entity linked via *specializationOf*, and token-related provenance (token entity, generation activity, Trusted Party agent). All nodes are persisted in Neo4j and linked to the meta-component. The method returns a boolean indicating whether a new meta-component was created or an existing one was reused. This value is exposed as the `metaComponentCreated` field in the store response.

For updates, it locates the current version entity, increments the version number, creates a new version entity linked to the previous one via *wasRevisionOf*, and adds token provenance for the new version.

Retrieval of meta-component is handled by `MetaService`, which reconstructs a full PROV document from the stored Neo4j graph. It queries all nodes and relationships belonging to the meta-component bundle using Cypher, then assembles them into a ProvToolbox `Document` object using `ProvFactory`. The result is serialized to the requested output format.

3.3.6 Subgraph Generation

The `SubgraphUtils` class implements the separation of a stored bundle into its traversal information (backbone) and domain-specific parts.

It classifies each element in the bundle based on CPM type attributes. Elements with CPM types (main activity, connectors, sender and receiver agents) and their direct relations form the backbone. All remaining elements form the domain-specific subgraph. Specialization relations that link domain-specific entities to backbone connectors are included in both subgraphs to preserve the connection between the two parts.

The generated subgraph is serialized to the requested output format and cached as a separate `Document` node in Neo4j to avoid repeated computation on subsequent retrievals.

3.3.7 Trusted Party Communication

Inter-service communication is implemented in the `TrustedPartyClient` class, which uses Spring's `RestTemplate` for HTTP calls to the Trusted Party. It provides methods for organization registration, signature verification, and token issuance.

The target Trusted Party URL is resolved per-request from the organization's registered configuration. If no organization-specific URL is available, the default endpoint from `AppProperties` is used. When Trusted Party integration is disabled (`app.disableTrustedParty=true`), the client is not invoked and a dummy token with placeholder values is used for meta-component construction.

3.3.8 Document Store Flow

To illustrate how the individual components interact, the following sequence describes the full processing of a POST store-document request:

1. `DocumentRestController` receives the HTTP request and delegates to the facade layer.
2. `DocumentFacadeImpl` passes the request to `DocumentService`.
3. `DocumentService.storeDocument()` orchestrates the remaining steps.
4. `InputGraphChecker.parseGraph()` parses the base64-encoded document, extracts the bundle, main activity, and connectors.

5. `InputGraphChecker.checkIdsMatch()` verifies that the bundle identifier matches the request path.

6. `InputGraphChecker.validateGraph()` runs CPM and PROV validation.

7. `ConnectorResolvabilityChecker.checkResolvability()` verifies that backward connectors and specialized forward connectors reference bundles that are reachable at their declared URIs. For each connector, the checker extracts the `cpm:referencedBundleId` and `cpm:referencedMetaBundleId` attributes and sends an HTTP HEAD request to the corresponding endpoint. If the remote server does not respond or returns a non-success status, the document is rejected.

8. The service checks whether a document with the same identifier already exists (conflict detection).

9. `TrustedPartyClient.verifySignature()` sends the document and signature to the Trusted Party for verification.

10. `TrustedPartyClient.issueToken()` requests a non-repudiation token for the verified document.

11. `ImportGraphService.importGraph()` persists the PROV graph nodes and relationships in Neo4j and creates the meta-component with version and token entities.

12. The token metadata is stored as a separate node for later retrieval.

Update requests follow the same sequence but skip the conflict check and instead locate the existing document to create a new version in the meta-component chain.

3.4 Trusted Party Service

3.4.1 REST Endpoints

The Trusted Party Service exposes its API through four controller classes. Table 3.2 lists all endpoints. All paths share the common prefix `/api/v1/`.

Table 3.2: REST endpoints of the Trusted Party Service.

URI pattern	Method	Purpose
`/info`	GET	Service identity and certificate
`/organizations`	GET	List all organizations
`/organizations/{orgId}`	GET	Get organization details
`/organizations/{orgId}`	POST	Register organization
`/organizations/{orgId}/certs`	GET	Get certificates
`/organizations/{orgId}/certs`	PUT	Update certificates
`/verifySignature`	POST	Verify document signature
`/issueToken`	POST	Issue non-repudiation token
`/organizations/{orgId}/tokens/{docId}/{format}`	GET	Get token for document
`/organizations/{orgId}/tokens`	GET	Get all tokens for org
`/organizations/{orgId}/documents/{docId}/{format}`	GET	Get stored document

The `issueToken` endpoint accepts a JSON body with fields: `organizationId`, `document` (base64), `documentFormat`, `documentType`, `createdOn`, and `signature` (required for GRAPH type). The document type determines what kind of provenance artifact the token covers:

- `GRAPH` – the full provenance document as submitted by a client organization, accompanied by a digital signature.
- `DOMAIN_SPECIFIC` – the domain-specific subgraph extracted from a stored document.
- `BACKBONE` – the traversal information subgraph.
- `META` – a meta-component recording version history.

Only GRAPH-type requests carry a client signature and require signature verification. The remaining types are requested internally by the Provenance Storage Service without a client signature.

3.4.2 Data Model

The Trusted Party data model uses JPA entities mapped to PostgreSQL:

- `Organization` – identified by organization name.
- `Certificate` – stores PEM-encoded certificate text, type (CLIENT or INTERMEDIATE), revocation status, and a SHA-256 digest of the DER encoding for deduplication. Linked to an organization.
- `Document` – stores the base64 document, format, type (GRAPH, DOMAIN_SPECIFIC, BACKBONE, META), creation timestamp, and signature. Linked to the submitting organization and the certificate active at submission time.
- `Token` – stores the document digest, hash function identifier, creation timestamp, and the Trusted Party signature. Linked to the document it was issued for.

3.4.3 Certificate Verification

Certificate chain verification is implemented in `TrustedPartyUtils.verifyChainOfTrust()`. The method constructs a PKIX `CertPath` from the submitted client and intermediate certificates and validates it against trust anchors loaded from a configured directory of root certificates.

On certificate update, the service revokes all previously active certificates for the organization and stores the new validated certificates. Intermediate certificates are deduplicated by their SHA-256 digest. The certificate history is retained for evidence purposes.

In the current implementation, CRL[7] and OCSP[8] revocation checking is disabled in the PKIX validation path.

7. Certificate Revocation List – a periodically published list of revoked certificates.
8. Online Certificate Status Protocol – a protocol for real-time certificate revocation checking.

3.4.4 Token Construction and Signing

Token issuance is implemented in `TokenService.issueToken()`. The process follows these steps:

1. Validates request fields and resolves the organization.
2. For GRAPH-type requests, verifies the submitted signature against the organization's active client certificate using `SHA256withECDSA`.
3. Parses the submitted document via ProvToolbox and resolves the bundle identifier.
4. Checks if a token already exists for the same document. If so, returns the existing token (idempotent behavior).
5. Persists the document record with its metadata and certificate reference.
6. Computes a SHA-256 digest of the decoded document bytes.
7. Assembles token data fields (originator, authority, timestamps, digest, additional data).
8. Canonicalizes the token data by recursively sorting JSON keys and serializing to UTF-8 bytes.
9. Signs the canonicalized payload using the Trusted Party private key with ECDSA/SHA-256 via Bouncy Castle.
10. Persists and returns the signed token.

For META-type requests, signature verification is skipped and the token is returned without persisting a document-token pair.

The timestamp parsing logic (`parseCreatedOn`) accepts both ISO-8601 strings and epoch seconds (as integers or milliseconds), providing backwards compatibility with the predecessor API format while supporting the new string-based format.

3.4.5 Signature Verification

Signature verification is exposed as a standalone operation. The service loads the active (non-revoked) client certificate for the specified organization, extracts its public key, and verifies the ECDSA signature over the base64-decoded document bytes. This operation is called by the Provenance Storage Service before token issuance in trusted-party-enabled mode.

3.4.6 Evidence Persistence

For GRAPH-type token issuance, the service persists a complete evidence chain: organization, active certificate at issuance time, document record (content, format, type, creation time, signature), and the issued token (digest, hash function, timestamp, TP signature). This allows later reconstruction of the full issuance context for dispute resolution.

For META-type issuance, the signed token is returned to the caller but no document-token pair is persisted. This asymmetry is noted as a limitation.

3.5 Error Handling

Both services use a centralized error handling mechanism implemented via Spring's `@RestControllerAdvice`. A global exception handler class intercepts exceptions thrown during request processing and maps them to structured JSON error responses. Each response contains a timestamp, HTTP status code, error message, and the request path.

Table 3.3 shows how domain exceptions are mapped to HTTP status codes in the Provenance Storage Service.

Table 3.3: Exception-to-HTTP-status mapping in the Provenance Storage Service.

Exception	Status	Meaning
`BadRequestException`	400	Invalid input or failed validation
`UnauthorizedException`	401	Signature verification failed
`NotFoundException`	404	Document or resource not found
`ConflictException`	409	Document already exists
`TrustedPartyUnavailable-Exception`	502	Trusted Party unreachable
`TrustedPartyDisabled-Exception`	503	Operation requires TP but it is disabled

The Trusted Party Service uses the same pattern with domain-specific exceptions:

- Not-found exceptions (organization, certificate, document) map to 404.
- Validation exceptions (invalid timestamp, missing signature, organization ID mismatch) map to 400.
- `OrganizationAlreadyExistsException` maps to 409.
- Verification failures (certificate chain, signature) map to 401.

This approach keeps error handling separated from business logic. Service classes throw domain exceptions, and the global handler translates them into consistent API responses without requiring error-handling code in controllers.

3.6 Configuration and Deployment

Both services use Spring Boot externalized configuration. All properties are defined in `application.yaml` with `${ENV_VAR}` placeholders that are resolved from environment variables at startup. Configuration

correctness follows standard Spring Boot behavior: database and service connections are validated on first use, not at startup. The Trusted Party Service additionally validates certificate file paths at startup via `@PostConstruct` and fails immediately if a certificate cannot be loaded.

3.6.1 Provenance Storage Service

Key configuration properties (bound in `AppProperties` via `@ConfigurationProperties`):

- `app.id` – service instance identifier, used as the originator identity when requesting meta-component tokens from the Trusted Party.
- `app.fqdn` – hostname and port of this service instance (e.g., `prov-storage-pathology:8000`). Used during validation to verify that the `referencedMetaBundleId` URI in submitted documents points to this server, and as the namespace base when generating meta-component PROV documents.
- `app.tpFqdn` – default Trusted Party endpoint.
- `app.disableTrustedParty` – boolean controlling operating mode.

Neo4j connection properties (`spring.neo4j.uri`, `spring.neo4j.authentication.username`, `spring.neo4j.authentication.password`) are standard Spring Data Neo4j properties, configured through the same mechanism.

3.6.2 Trusted Party Service

Key configuration properties:

- `app.id` – Trusted Party authority identifier (included in token data).
- `app.fqdn` – Trusted Party address (included in token metadata).
- `app.trustedCertsDirPath` – directory of trusted root certificates for PKIX validation.

- `app.publicCertPath` – Trusted Party public certificate (exposed in info endpoint and token metadata).
- `app.privateKeyPath` – Trusted Party private key for ECDSA token signing.

PostgreSQL connection uses standard Spring datasource properties (`spring.datasource.*`).

3.6.3 Docker Compose Deployment

The `docker-compose.yml` defines the full deployment topology: Neo4j, PostgreSQL, both Spring Boot services, Prometheus, and Grafana. Services communicate over the Docker network using service hostnames.

For simplicity, the development setup uses plain HTTP for inter-service communication within the isolated Docker network. Switching to HTTPS in production requires only deploying the Trusted Party Service with a TLS certificate and updating the base URL configuration, with no changes to the application code.

Certificate material is mounted as a read-only volume. Each service is built from a module-specific multi-stage Dockerfile.

Published ports for local access: Provenance Storage API (8000), Trusted Party API (8020), Neo4j Bolt (7687) and browser (7474), PostgreSQL (5432), Prometheus (9090), Grafana (3000).

Both services expose OpenAPI documentation via Swagger UI at `/swagger-ui/index.html`.

Once the certificate material is in place and environment variables are configured, the entire stack is started with a single command:

```
docker compose up
```

For local development without Docker, individual services can be started using the Maven Spring Boot plugin:

```
./mvnw -pl distributed-prov-system-service spring-boot:run
./mvnw -pl trusted-party-service spring-boot:run
```

3.6.4 Monitoring

Both services include Spring Boot Actuator with a Micrometer Prometheus registry. Each service exposes a `/actuator/prometheus` endpoint that

publishes JVM metrics (memory, threads, garbage collection), HTTP request metrics (count, duration, status), and database connection pool statistics. The Docker Compose deployment includes a Prometheus instance configured to scrape both services and a Grafana instance for visualization.

3.7 Limitations

- **No PKIX revocation checking.** Certificate chain validation does not enforce CRL or OCSP checks.
- **Subgraph semantic validation not implemented.** When the Trusted Party issues a token for a DOMAIN_SPECIFIC or BACKBONE document, it should verify that the submitted subgraph is a valid subset of the original stored document. The method `checkIsSubgraph` is called at the correct point in the token issuance flow, but its body is empty and the original bundle is not retrieved from the Provenance Storage Service. This feature was also left unimplemented in the Python predecessor.
- **Asymmetric evidence persistence.** META-type token issuance does not persist a document-token pair, unlike GRAPH-type issuance.
- **Not evaluated under production load.** Performance, high availability, and production secret management are outside the current scope.

4 Tests

This chapter describes how the implementation is validated through automated tests. The test suite contains approximately 328 test methods across 44 test classes, organized in a layered structure that mirrors the application architecture. All tests can be executed with a single command from the repository root:

```
./mvnw test
```

Individual service modules can be tested separately using the `-pl` flag (e.g., `./mvnw -pl trusted-party-service test`).

4.1 Test Frameworks and Infrastructure

The test suite uses the following frameworks:

- **JUnit 5** as the test runner and lifecycle manager.
- **Mockito** for mocking dependencies, including `@Mock`, `@MockitoBean`, `MockedConstruction`, and `spy` patterns for isolating control-flow behavior.
- **AssertJ** for fluent assertions (`assertThat`, `assertThatThrownBy`).
- **Spring Boot Test** for slice and integration testing (`@WebMvcTest`, `@DataNeo4jTest`, `@DataJpaTest`, `@SpringBootTest`).
- **Neo4j Harness** for embedded graph database instances in repository tests.

4.1.1 Neo4j Test Harness

Repository tests in the Provenance Storage Service require a running Neo4j instance. Instead of an external database, the tests use Neo4j Harness, which starts an in-process Neo4j instance with the Bolt protocol disabled. An abstract base class `AbstractNeo4jRepositoryTest` configures the embedded instance using `@DynamicPropertySource` to inject the Bolt URI into the Spring context. Each test method starts with a clean database state by executing `MATCH (n) DETACH DELETE n` in a `@BeforeEach` method.

4.1.2 Test Data Factory

Both services include a `TestDataFactory` class that provides static helper methods for constructing predefined test fixtures. The Provenance Storage Service version is particularly notable because it assembles CPM-compliant PROV-JSON documents programmatically. The factory method `minimalCpmProvJson()` constructs a valid provenance bundle with correct namespace URIs, a main activity, a current agent, forward connectors, and meta-bundle references. This allows component tests to submit structurally valid documents without maintaining separate test resource files.

The Trusted Party `TestDataFactory` provides request DTOs for token issuance, organization registration, and certificate updates.

The Trusted Party `TestDataFactory` generates request DTOs for token issuance, organization registration, and certificate updates.

4.1.3 Test Profiles

The Trusted Party component tests run under a dedicated Spring profile (`componenttest`) configured in `application-componenttest.yaml`. This profile uses H2 (an in-memory database) in PostgreSQL compatibility mode, eliminating the need for an external database. Test certificate material is stored in a `certs/` directory within the test resources.

4.2 Unit Tests

Unit tests target isolated business logic in services, facades, mappers, and utility classes. Dependencies are mocked to keep execution deterministic and focused on single-method behavior.

Tests follow a consistent naming convention: `MethodName_StateUnderTest_ExpectedBehavior`. Test fixtures are prepared in `@BeforeEach` methods to ensure deterministic initial state before each test case.

4.2.1 Provenance Storage Service

Unit tests validate document, organization, and meta-component processing logic. Key test classes include:

- `DocumentServiceTest` – tests document storage and retrieval with both TP-enabled and TP-disabled control branches, validation failures, conflict detection, and exception paths. Uses `MockedConstruction` to isolate the `InputGraphChecker` pipeline.
- `ImportGraphServiceTest` – tests meta-component creation and version-chain updates using mocked repositories.
- `MetaServiceTest` – tests PROV document reconstruction from stored graph nodes.
- `OrganizationServiceTest` – tests organization registration and Trusted Party forwarding logic.

Facade tests (`DocumentFacadeImplTest`, `MetaFacadeImplTest`, `OrganizationFacadeImplTest`) verify that facade methods correctly delegate to service methods and propagate exceptions.

4.2.2 Trusted Party Service

Unit tests validate certificate processing, token request handling, signature-related branches, and repository interaction. Key test classes include:

- `TokenServiceTest` – tests token issuance flow including signature verification, idempotent behavior (returning existing tokens), timestamp parsing for both ISO-8601 and epoch formats, and error paths. Uses `spy` patterns to isolate control-flow behavior while allowing partial real execution.
- `OrganizationServiceTest` – tests organization registration, certificate chain processing, and revocation logic.
- `DocumentServiceTest` – tests document retrieval and format filtering.
- `InfoServiceTest` – tests service identity endpoint logic.

4.2.3 Repository Slice Tests

Repository behavior is validated using Spring Data slice tests[1]. The Provenance Storage Service uses `@DataNeo4jTest` with the Neo4j Harness described above. These tests cover CRUD operations on all PROV node types (`Activity`, `Agent`, `Bundle`, `Entity`, `Document`, `Token`). The Trusted Party Service uses `@DataJpaTest` with H2[2] to verify JPA entity mappings and custom query methods.

4.2.4 Utility Tests

`InputGraphCheckerIdMatchingTest` validates the bundle ID matching and URI path resolution logic in `InputGraphChecker`. It uses real PROV-JSON documents from `TestDataFactory` to test that bundle identifiers are correctly extracted and matched against request paths. CPM and PROV validators are mocked in these tests to focus specifically on the ID resolution logic.

`TrustedPartyUtilsTest` validates certificate-related helper methods. Tests that require real certificate material are conditionally skipped when the files are unavailable, since private keys are not committed to the repository.

4.3 Component and Integration Tests

Component tests validate API-level behavior and HTTP contracts by exercising the controller layer with either mocked or real downstream dependencies.

4.3.1 Provenance Storage Service

API component tests use `@WebMvcTest` with `@MockitoBean`-annotated collaborators that mock service and repository dependencies. This

1. Slice tests load only the Spring context relevant to a specific layer (e.g., only the persistence layer), rather than the full application context. This makes them faster and more focused than full integration tests.
2. H2 is a lightweight in-memory Java SQL database commonly used for testing. It can emulate PostgreSQL dialect, allowing tests to run without an external database server.

setup tests controller endpoints, request validation, HTTP status mapping, and exception translation in isolation. Three test classes cover the main API areas:

- `DocumentApiComponentTest` – document store, update, retrieve, and subgraph endpoints.
- `MetaApiComponentTest` – meta-component retrieval and existence checks.
- `OrganizationApiComponentTest` – organization registration and updates.

REST controller tests (`DocumentRestControllerTest`, `MetaRestControllerTest`, `OrganizationRestControllerTest`) provide additional endpoint coverage with emphasis on mode-dependent control flow (TP-enabled versus TP-disabled).

4.3.2 Trusted Party Service

The Trusted Party component test class (`TrustedPartyApiComponentTest`) uses `@SpringBootTest(webEnvironment = RANDOM_PORT)` with the `componenttest` profile. Unlike the Provenance Storage Service tests, requests here traverse the full runtime path from controller through service and repository layers to the test database. This provides true integration coverage.

Verified scenarios include organization registration, certificate updates, token issuance with real signature verification, document retrieval, and all major exception paths. These tests contributed to implementation hardening by exposing defects such as global exception-handler wiring issues and null-handling edge cases that were corrected during development.

4.4 Test Coverage Summary

Table 4.1 summarizes the test distribution across both services.

Table 4.1: Test suite structure and coverage areas.

Test layer	Count	What is validated
Service unit tests	13 classes	Business logic, control branches, exception paths
Facade unit tests	7 classes	Delegation correctness
Repository slice tests	13 classes	CRUD operations, query methods, entity mappings
Utility tests	3 classes	ID resolution, certificate helpers
API component tests (mocked)	6 classes	HTTP contracts, status codes, request validation
Full integration tests	1 class	End-to-end request flow with real database

4.5 Limitations of the Test Suite

The `CPMValidatorImpl`, `SubgraphUtils`, and `ProvToolboxUtils` classes do not have dedicated unit tests. These components were validated manually during development through Docker-based integration runs with PROV documents. Their correctness depends on the ProvToolbox library behaving as documented and on the CPM specification being correctly translated into validation rules. Adding dedicated unit tests for these classes would improve regression safety and is noted as future work.

Additionally, the Provenance Storage Service lacks a full integration test that exercises the complete request flow from controller to database. The Trusted Party Service has `TrustedPartyApiComponentTest`, which uses `@SpringBootTest` with a random port and an H2 in-memory database. This test starts the full Spring application context, sends HTTP requests via `TestRestTemplate`, and verifies the entire path through all layers: controller, facade, service, repository, and database. The test uses plain HTTP because TLS is not configured at the application level and is outside the scope of this project. In contrast, the Provenance Storage Service only has `@WebMvcTest`-based tests that mock all service and repository dependencies, verifying HTTP contracts and request validation but not catching issues in the interaction between layers.

5 Future Work

The current implementation fulfills the thesis requirements but several areas remain for future improvement.

- **Certificate revocation checking.** The PKIX certificate chain validation currently does not check whether certificates have been revoked. This limitation was also present in the predecessor Python implementation, which uses pyOpenSSL `X509StoreContext` without enabling CRL flags. Implementing revocation checking would require network access to CRL distribution points or OCSP responders, timeout and caching strategies, and fallback behavior when these services are unavailable. Adding CRL and OCSP support would allow the system to reject signatures made with compromised certificates.

- **Semantic subgraph validation.** The system can split a provenance document into its domain-specific and backbone (traversal) parts. The splitting algorithm works as follows: it first separates all bundle statements into elements (entities, activities, agents) and relations. Each element is classified as backbone if it carries a CPM type (e.g. `cpm:forwardConnector`, `cpm:mainActivity`) and contains no non-backbone attributes (attributes outside the CPM and PROV namespaces). Specialized forward connectors (entities that specialize a general forward connector) are also classified as backbone. All remaining elements are classified as domain-specific. Relations are assigned to the backbone if both their source and target elements belong to the backbone set, otherwise they are placed in the domain-specific part.

  However, the system does not verify that a separately submitted subgraph is a valid subset of an existing document. The algorithm could misclassify elements if a domain-specific entity accidentally carries only CPM-namespace attributes, or if a relation references an element not present in the current bundle. Implementing a validation step that checks whether a submitted subgraph is structurally consistent with the full stored document would prevent inconsistent partial submissions.

- **Dedicated unit tests for validation utilities.**
  The `CPMValidatorImpl`, `SubgraphUtils`, and `ProvToolboxUtils` classes lack dedicated unit tests. Adding them would improve regression safety for the core validation logic.

- **Full integration test for the Provenance Storage Service.** The existing `DocumentApiComponentTest` uses `@WebMvcTest` which only loads the controller layer and mocks all service and repository dependencies. This verifies HTTP request handling and response mapping but does not test the interaction between the service, repository, and Neo4j database layers. Unlike the Trusted Party Service, where H2 can substitute for PostgreSQL in tests, Neo4j is a graph database that cannot be replaced by an in-memory relational substitute. A full integration test would require either Testcontainers with a Neo4j Docker image or an embedded Neo4j instance, adding infrastructure complexity to the test suite.

- **Performance evaluation.** The system has not been benchmarked under production-scale load. Measuring throughput and latency for document storage, token issuance, and graph traversal under concurrency would identify bottlenecks.

- **Production deployment hardening.** The Docker Compose setup is suitable for development and demonstration. A production deployment would require proper secret management, high-availability configuration, and health-based alerting.

6 Conclusion

The goal of this thesis was to rewrite the Python-based Provenance Storage System into Java while preserving its functionality. The system had to support CRUD operations for provenance documents, maintain non-repudiation evidence, and enable authenticity verification at three levels of granularity. The implementation had to be modular and easily extensible.

All of these requirements were fulfilled. The resulting system consists of two Spring Boot microservices (Provenance Storage Service and Trusted Party Service) that together provide the required functionality:

1. **CRUD operations.** The Provenance Storage Service supports storing, updating, retrieving, and checking existence of provenance documents. Documents are validated against both W3C PROV constraints and Common Provenance Model rules before persistence.

2. **Non-repudiation evidence.** For each stored document, the Trusted Party Service issues a signed non-repudiation token containing a document digest, timestamps, and authority information. The token is signed with ECDSA and persisted together with the original document, submitter certificate, and signature, forming a complete evidence chain.

3. **Authenticity verification.** The system verifies document authenticity by validating the submitter's digital signature against their registered certificate chain before accepting a document for storage. The Trusted Party acts as a neutral authority that can confirm document origin in case of disputes.

4. **Three levels of granularity.** Retrieval and verification are supported for the full provenance document, its domain-specific part, and its traversal information (backbone). Each level has its own non-repudiation token.

5. **Modularity.** The system uses a layered architecture (controller, facade, service, repository) with clear separation of concerns.

The two services communicate through a REST API and can be developed, deployed, and scaled independently. The Trusted Party can be disabled for environments that do not require non-repudiation.

The implementation was validated through automated tests covering service logic, repository operations, and API contracts. Known limitations (disabled certificate revocation checks and incomplete subgraph semantic validation) are documented and identified as future work.

The source code is publicly available on GitHub under MIT license and the system can be deployed using a single `docker compose up` command with the provided configuration.

Bibliography

1. GROTH, Paul; MOREAU, Luc. *PROV-Overview: An Overview of the PROV Family of Documents* [online]. World Wide Web Consortium (W3C), 2013-04. [visited on 2026-03-21]. Available from: https://www.w3.org/TR/prov-overview/.
2. MOREAU, Luc; MISSIER, Paolo. *PROV-DM: The PROV Data Model* [online]. World Wide Web Consortium (W3C), 2013-04. [visited on 2026-03-21]. Available from: https://www.w3.org/TR/prov-dm/.
3. WITTNER, Rudolf; GALLO, Matej; FREXIA, Francesca; LEO, Simone; PIREDDU, Luca; MASCIA, Cecilia; PLASS, Markus; EKLUND, Niina; SOILAND-REYES, Stian; MÜLLER, Heimo; GEIGER, Joerg. *Common Provenance Framework for Multi-organizational Environments* [online]. Zenodo, 2025 [visited on 2026-05-13]. Available from DOI: 10.5281/zenodo.18063102.
4. GAJDOŠ, David. *Implementation of Integrity and Non-repudiation for Distributed Provenance Information* [*online*] [online]. 2024. [visited on 2026-03-21]. Available from: https://is.muni.cz/th/mo8f1/. Diplomová práce. Masarykova univerzita, Fakulta informatiky, Brno. SUPERVISOR : Rudolf Wittner.
5. MENEZES, Alfred J.; OORSCHOT, Paul C. van; VANSTONE, Scott A. *Handbook of Applied Cryptography*. CRC Press, 1996. ISBN 0849385237.
6. MOJŽIŠ, Martin. *Unit and Component Tests of a Storage System* [*online*] [online]. 2025. [visited on 2026-03-21]. Available from: https://is.muni.cz/th/du7ec/. Diplomová práce. Masarykova univerzita, Fakulta informatiky, Brno. SUPERVISOR : Rudolf Wittner.
7. MOREAU, Luc; MISSIER, Paolo. *PROV-N: The Provenance Notation* [online]. World Wide Web Consortium (W3C), 2013-04. [visited on 2026-03-21]. Available from: https://www.w3.org/TR/prov-n/.

8. LEBO, Timothy; SAHOO, Satya; MCGUINNESS, Deborah. *PROV-O: The PROV Ontology* [online]. World Wide Web Consortium (W3C), 2013-04. [visited on 2026-03-21]. Available from: `https://www.w3.org/TR/prov-o/`.

9. *ISO/IEC 13888-1: Information Technology – Security Techniques – Non-repudiation – Part 1: General*. International Organization for Standardization, 2000. First edition.

10. CHENEY, James; MISSIER, Paolo; MOREAU, Luc. *Constraints of the PROV Data Model* [online]. World Wide Web Consortium (W3C), 2013-04. [visited on 2026-03-21]. Available from: `https://www.w3.org/TR/prov-constraints/`.

A Electronic Archive Structure

The repository is organized as follows:

```
provenance-storage-java/
  README.md                       -- Project overview and deployment guide
  pom.xml                         -- Parent Maven POM
  mvnw                            -- Maven wrapper script
  docker-compose.yml              -- Full deployment definition
  certs/
    cert-dev/
      cert.pem                    -- Development certificate
      ec_key.pem                  -- Development private key
    trusted_certs/
      ca.pem                      -- Root CA certificate
  prometheus/
    prometheus.yml                -- Scrape configuration
  grafana/
    provisioning/                 -- Datasource definitions
    dashboards/                   -- Pre-configured dashboards
  distributed-prov-system-service/
    Dockerfile
    pom.xml
    src/main/java/.../
      rest/                       -- REST controllers
      facade/                     -- Facade layer
      service/                    -- Business logic
      data/model/nodes/           -- Neo4j node entities
      data/model/relationships/   -- Neo4j relationship entities
      data/repository/            -- Spring Data Neo4j repos
      client/                     -- Trusted Party REST client
      utils/prov/                 -- Validation utilities
      api/                        -- Request/response DTOs
      config/                     -- AppProperties
    src/main/resources/
      application.yaml            -- Service configuration
    src/test/                     -- Unit and component tests
  trusted-party-service/
```

```
Dockerfile
pom.xml
src/main/java/.../
  rest/                       -- REST controllers
  facade/                     -- Facade layer
  service/                    -- Business logic
  data/model/                 -- JPA entities
  data/repository/            -- Spring Data JPA repos
  mappers/                    -- DTO-to-entity mappers
  utils/                      -- Crypto and PROV utilities
  api/                        -- Request/response DTOs
  config/                     -- AppProperties
src/main/resources/
  application.yaml            -- Service configuration
src/test/                     -- Unit and component tests
```

Instructions for building, configuring, and deploying the application are provided in the `README.md` file at the root of the repository.

The source code is available at `https://github.com/xkroslak/provenance-storage-java`.