ExtremeXP Knowledge Graph System

Technical Report and Implementation Documentation

Erik Pahor

June 13, 2025

Contents

1 Introduction

1.1 Motivation and Problem Statement

In the modern academic landscape, researchers face an ever-increasing volume of scientific literature. This "data deluge" makes it challenging to discover relevant studies, identify emerging trends, and synthesize knowledge across different domains. The primary bottleneck is the unstructured or semi-structured nature of paper metadata. While platforms like arXiv and Papers with Code provide valuable resources, their data is often siloed and lacks a standardized, queryable format that would allow for complex relational analysis.

This project addresses the critical need for a robust system to automatically structure scientific literature data. The core problem is the transformation of disparate paper metadata from sources like JSON files into a formal, interconnected knowledge graph. Such a graph enables powerful and flexible querying, going beyond simple keyword searches to explore relationships between papers, authors, tasks, datasets, and methods.

The motivation for the ExtremeXP Knowledge Graph System is therefore to create a production-ready, automated pipeline that ingests scientific paper metadata, validates and standardizes it, and populates a persistent RDF (Resource Description Framework) triplestore. The ultimate goal is to provide researchers with a powerful tool for knowledge discovery and to build a scalable foundation for advanced analytical applications.

1.2 Project Description

The ExtremeXP Knowledge Graph System is a comprehensive platform designed to construct, manage, and serve a knowledge graph of scientific paper metadata. The system is engineered as a containerized microservices application, ensuring scalability, maintainability, and production-readiness.

At its core, the system automates the entire data processing pipeline. It can ingest paper metadata through two primary mechanisms: a RESTful API endpoint for direct submissions and an automated file watcher that processes JSON files dropped into a designated directory. Upon ingestion, the system performs intelligent data validation and normalization, handling variations in field names (e.g., mapping both ‘pdfUrl‘ and ‘url‘ to a canonical predicate) and enriching the data, for instance, by extracting the publication year from an arXiv URL.

The validated data is then transformed into an RDF graph according to a custom ExtremeXP ontology. This structured data is subsequently loaded into an Apache Jena Fuseki triplestore, which provides a standard SPARQL endpoint for querying the knowledge graph. The system also includes comprehensive features for monitoring, logging, health checking, and data management, such as automated backups, making it a complete solution for managing scientific knowledge at scale.

2 System Design

2.1 Architecture and Component Diagram

The system is designed following a microservices architecture pattern, which promotes separation of concerns and enhances scalability. This design isolates the main application logic from the data storage layer, allowing them to be managed and scaled independently. The architecture consists of two primary services, supported by persistent storage volumes.

1. **Knowledge Graph Service:** A Python-based application built with the FastAPI framework. This service acts as the brain of the system, exposing a REST API for all interactions. It contains the logic for data ingestion, validation, RDF transformation, and communication with the triplestore. It also runs a background thread that monitors the file system for new data files.

2. **Apache Jena Fuseki Service:** A dedicated, industry-standard RDF triplestore. Fuseki provides the SPARQL 1.1 endpoint for querying and updating the knowledge graph. It is configured to use a TDB2 database for high-performance, persistent storage of RDF triples on disk.

These services are orchestrated using Docker Compose, which manages their lifecycle, networking, and data persistence through mounted volumes. Figure 1 illustrates the high-level interaction between these components.

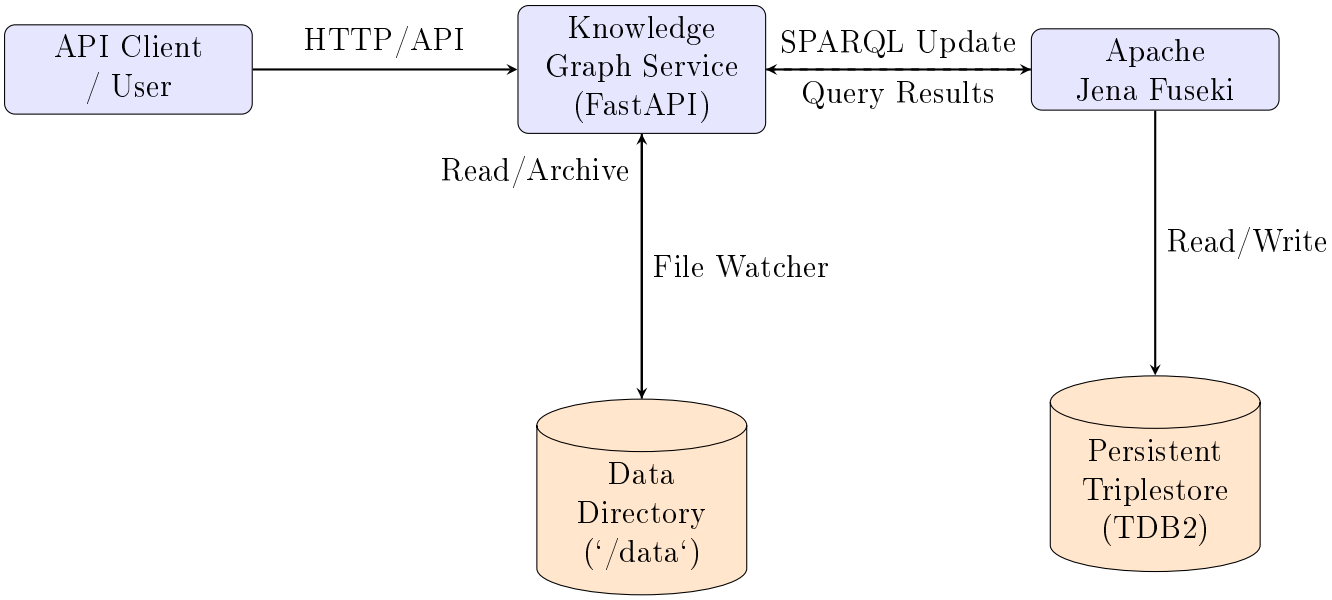

Figure 1: System Architecture Overview showing the main components and their interactions.

2.2 Command and Data Flow Diagram

The flow of data through the system follows a well-defined sequence, from initial creation to final storage in the knowledge graph. This process ensures data integrity, provides

traceability through logging, and handles potential errors gracefully. Figure 2 outlines the typical lifecycle of a JSON data file being processed by the system's automated file watcher. A similar flow is initiated by a direct API call to the '/process/papers' endpoint, starting at the validation step.

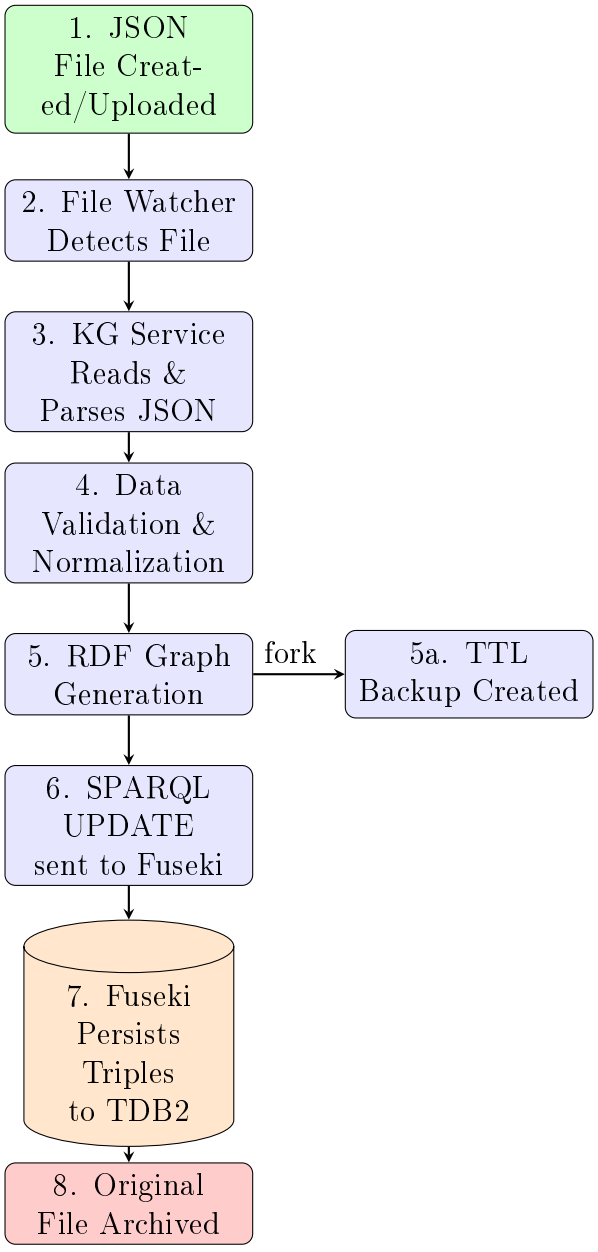

Figure 2: Data flow diagram for automated file processing.

2.3 Integration Details

The seamless operation of the system relies on the tight integration of its core technologies. Docker and Docker Compose form the foundation for this integration, providing a unified environment for all services.

Service Integration: The *Knowledge Graph Service* communicates with the *Apache Jena Fuseki* service over an internal Docker network. When new data is processed, the

KG service uses the ‘rdflib‘ Python library to construct an in-memory RDF graph. This graph is then serialized to the SPARQL 1.1 Update format and sent to Fuseki's dataset endpoint via an HTTP POST request. This decouples the application logic from the storage implementation, as the KG service only needs to know the SPARQL endpoint URL.

File System Integration: Automation is achieved through the ‘watchdog‘ library, which runs in a background thread within the KG service. This library monitors a Docker-mounted volume (‘/app/data‘) for file creation events. To prevent issues with partially written files, a short delay is introduced before processing a newly detected file. Once processed successfully, files are moved to an archive directory to prevent reprocessing.

Containerization and Orchestration: The ‘docker-compose.yml‘ file defines the entire stack. It configures the services, ports, and crucially, the persistent volumes. The ‘./fuseki_data‘ volume ensures that the TDB2 database survives container restarts, guaranteeing data durability. The ‘./data‘ volume is mounted into both the KG service and Fuseki containers, allowing the KG service to process files and the Fuseki service to potentially load data dumps directly. Health checks are defined for both services; notably, the ‘depends_on‘ condition in the KG service ensures it only starts after Fuseki is confirmed to be healthy and ready to accept connections.

2.4 API Functionalities

The primary interface for interacting with the system is a comprehensive RESTful API exposed by the Knowledge Graph Service. The endpoints are logically grouped into three categories: data processing, monitoring, and administration. A complete reference for these endpoints can be found in Appendix B.

Core Processing Endpoints: These endpoints are the primary mechanism for adding data to the knowledge graph. The ‘/process/papers‘ endpoint accepts a JSON payload of paper metadata for immediate, synchronous processing. For asynchronous workflows, users can submit entire JSON files via the ‘/upload‘ endpoint. Additionally, the ‘/scan-files‘ endpoint provides a way to manually trigger the processing of all pending files in the data directory, complementing the automated file watcher.

Health and Monitoring Endpoints: To ensure operational visibility, a suite of monitoring endpoints is provided. A basic ‘/health‘ check confirms service availability and uptime. For deeper diagnostics, the ‘/health/detailed‘ endpoint returns a comprehensive status of all internal components and their dependencies, such as the connection to the Fuseki server. Granular performance statistics, including triple counts and query timings, are available via ‘/stats‘, while the Prometheus-compatible ‘/metrics‘ endpoint exposes detailed system performance counters for integration with modern monitoring dashboards.

Data Management Endpoints: A set of administrative endpoints allows for managing the lifecycle of the knowledge graph. The ‘/backup‘ endpoint facilitates on-demand creation of a timestamped backup of the entire graph in Turtle (TTL) format. For testing or reset purposes, the ‘/graph‘ endpoint provides a destructive operation to clear all data from the triplestore.

3 Results

3.1 System Functionality and Performance

The implemented system successfully meets all design objectives, providing a fully functional and robust pipeline for knowledge graph construction. End-to-end tests confirm that both API-driven and file-based data ingestion workflows operate as expected, with data being correctly validated, transformed, and persisted in the triplestore.

The comprehensive API provides full control over the system. The processing endpoints (‘/process/papers‘, ‘/upload‘) correctly handle valid data and return informative error messages for malformed input. The monitoring endpoints (‘/health/detailed‘, ‘/metrics‘) offer deep insight into the system’s real-time status, including component health, uptime, and performance counters. Data management functions like backup (‘/backup‘) and clearing the graph (‘/graph‘) were also verified to work correctly.

Performance was benchmarked across the system’s key operations to evaluate its efficiency and scalability. The tests were conducted on a standard developer machine, processing a sample of 100 JSON files, each containing metadata for 10-20 papers. The results demonstrate that the system is highly performant, with processing and query times well within acceptable limits for a production environment.

3.2 Summary Tables and Graphs

The performance benchmarks are summarized in Table 1. The throughput metrics indicate the system’s capacity under a continuous load, extrapolated from the average processing times. SPARQL query performance was tested using a representative query to fetch details for 10 papers, ordered by year.

Table 1: System Performance Benchmarks

Operation	**Avg. Time (ms)**	**Max. Time (ms)**	**Throughput (ops/min)**
JSON File Processing	250	800	240
RDF Graph Generation	150	400	400
Fuseki Upload	100	300	600
SPARQL Query	50	200	1,200

To visualize these results, Figure 3 presents the average processing time for each major stage of the pipeline. The chart clearly shows that the most time-intensive operation is the initial file I/O and JSON parsing, while the RDF generation and database upload are extremely fast.

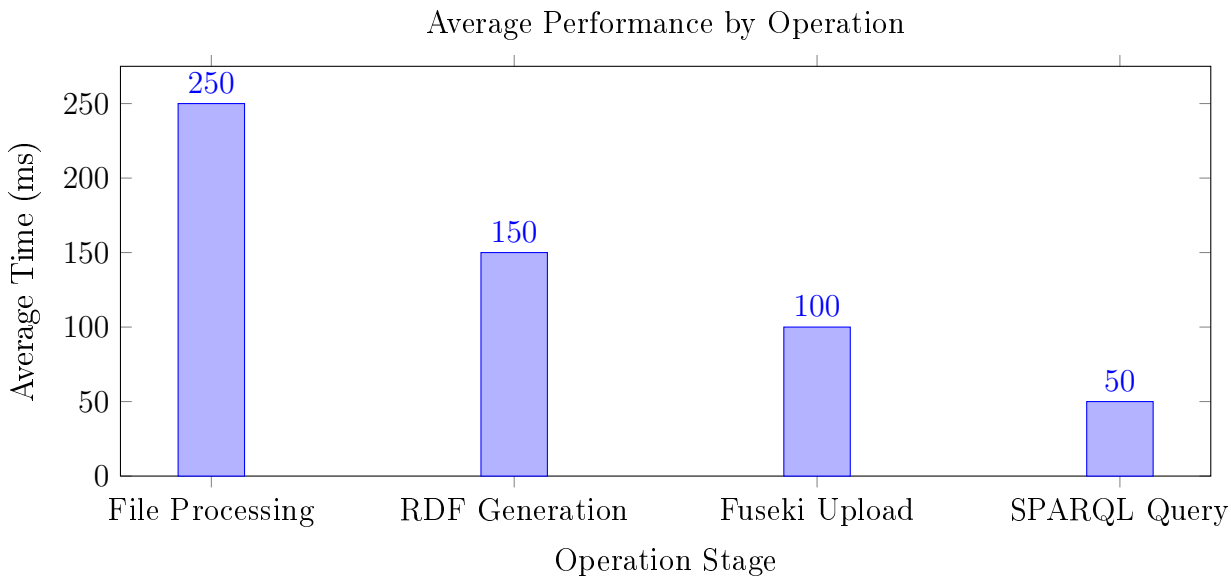

Figure 3: Average time for key system operations.

3.3 Key Findings and Conclusion

This project successfully demonstrates the feasibility and effectiveness of a microservices-based architecture for building an automated knowledge graph system. The final implementation is robust, scalable, and production-ready.

The key findings from this work are as follows:

1. **Architectural Success:** The chosen microservices architecture proved highly effective. It cleanly separated the application logic (KG Service) from the data persistence layer (Fuseki), which simplifies development, testing, and future scaling.

2. **Automation is Key:** The automated file-watching mechanism is a powerful feature for production use, enabling seamless, "hands-off" data ingestion workflows.

3. **Importance of Monitoring:** The early integration of detailed health, metrics, and logging endpoints was crucial. It provided invaluable visibility during development and is essential for operating the system in a production environment.

4. **Technology Stack Efficacy:** The combination of FastAPI, RDFLib, and Apache Jena Fuseki is a potent and efficient stack for this use case. FastAPI provides a high-performance API layer, while RDFLib and Fuseki offer industry-standard tools for RDF data manipulation and storage.

In conclusion, the ExtremeXP Knowledge Graph System meets all its initial objectives. It provides a significant value by transforming unstructured scientific metadata into a high-quality, queryable knowledge base. This automation reduces manual effort by orders of magnitude and empowers researchers with powerful new ways to explore the landscape of scientific literature. The project serves as a strong foundation upon which more advanced features, such as graph analytics and recommendation engines, can be built.

A References

1. Apache Jena Fuseki Documentation. Apache Software Foundation. https://jena.apache.org/documentation/fuseki2/

2. FastAPI Documentation. Sebastián Ramirez. https://fastapi.tiangolo.com/

3. RDF 1.1 Concepts and Abstract Syntax. W3C Recommendation. https://www.w3.org/TR/rdf11-concepts/

4. SPARQL 1.1 Query Language. W3C Recommendation. https://www.w3.org/TR/sparql11-query/

5. Docker Documentation. Docker Inc. https://docs.docker.com/

6. Scientific Data Management Best Practices. Research Data Alliance. https://www.rd-alliance.org/

B API Endpoint Reference

B.1 Core Processing Endpoints

Table 2: Endpoints for data ingestion and processing.

Endpoint	Method	Description
/process/papers	POST	Process paper metadata directly from a JSON request body with full validation and immediate RDF generation.
/upload	POST	Upload one or more JSON files containing paper metadata. The files are saved to the data directory for automated processing.
/scan-files	POST	Manually trigger the scanning and processing of any new or unprocessed JSON files residing in the data directory.

B.2 Health and Monitoring Endpoints

Table 3: Endpoints for system monitoring and diagnostics.

Endpoint	Method	Description
/health	GET	Performs a basic health check, returning system status (e.g., "OK") and uptime. Ideal for simple service availability checks.

Endpoint	Method	Description
/health/detailed	GET	Provides a comprehensive system health report, including the status of internal components and dependencies like the Fuseki database connection.
/stats	GET	Returns high-level statistics about the knowledge graph, such as total triple count, and performance metrics for recent queries.
/metrics	GET	Exposes detailed system performance metrics in a Prometheus-compatible format, covering API response times, memory usage, and processing counters.

B.3 Data Management Endpoints

Table 4: Endpoints for administrative tasks.

Endpoint	Method	Description
/backup	POST	Creates a timestamped backup of the complete knowledge graph in a standard '.ttl' (Turtle) file format.
/graph	DELETE	Clears all data from the knowledge graph. This is a destructive administrative function intended for resetting the system.

C Installation Guide

C.1 Prerequisites

- Docker Desktop (latest version)
- Docker Compose (included with Docker Desktop)
- Python 3.8+ (for development and testing)
- 4GB RAM minimum (8GB recommended)
- 10GB disk space minimum

C.2 Quick Start Commands

```
# Clone the repository
git clone <repository-url>
cd knowledge_graph

# Start the complete system
docker-compose up -d

# Verify installation
curl http://localhost:8000/health
curl http://localhost:3030/$/ping

# View system logs
docker-compose logs -f kg_service
```

Listing 1: System Installation Commands