ETD Collection Management

CS5604 Final Report - Team 1

CS5604 — Information Storage and Retrieval
Department of Computer Science
Virginia Polytechnic Institute and State University
Blacksburg, Virginia 24061

Team 1
Raghav Sethi
Ashrith Reddy Thukkaraju
Hirva Bhagat
Tanya Jain
Wen-yu Lee

SMEs: Prof. William A. Ingram and Prof. Sung Hee Park
Instructor: Dr. Edward Fox
January 13, 2023

Abstract

Academic institutions the world over are known to produce hundreds of thousands of ETDs (Electronic Theses and Dissertations) every year. At the end of an academic year, we are left with large volumes of ETD data that are rarely used for further research or ever cited in future work, writings, or publications. As part of the CS5604: Information Storage and Retrieval graduate-level course at Virginia Polytechnic Institute and State University (Virginia Tech), we collectively created a search engine for a collection of more than 500,000 ETDs from academic institutions in the United States, which constitutes the class-wide project. This system enables users to ingest, pre-process, and store ETDs in a repository; apply deep learning models to perform topic modeling, text segmentation, chapter summarization, and classification, backed by a DevOps, user experience and integrations team. We are Team 1 or the "ETD Collection Management" team. During the course of the Fall 2022 semester at Virginia Tech, we were responsible for setting up the repository of ETDs, which encompasses broadly the following three components: (1) setting up a database, (2) storing digital objects in a file system, and (3) creating a knowledge graph. Our work enabled other teams to efficiently retrieve the stored ETD data, and perform appropriate pre-processing operations, and during the final few months of the semester, to apply the aforementioned deep learning models to the ETD collection we created. The key deliverable for Team 1 was to create an interactive user interface to perform CRUD operations (create, retrieve, update, and delete) in order to interact with the repository of ETDs, which is essentially an extrapolation of the work already taken up at Virginia Tech's Digital Library Research Laboratory. Owing to the fact that the other teams had no direct access to the repository set up by us, we designed a host of Application Programming Interfaces (APIs) which are elaborated in depth in the subsequent sections of the document. The end goal for Team 1 was to be able to set up an accessible repository of ETDs so that they can be used for further research work. This is taking into account how each ETD is a well-curated resource and how it may even prove to be an excellent asset for an in-depth analysis on a certain topic, not limited to academic or research purposes.

Acknowledgments

First and foremost we would like to express our sincere gratitude to Professor Edward A. Fox and thank him for his continuous motivation and support. His valuable suggestions and insights were of great help to us. We thank him for offering continuous and invaluable guidance and for pushing us to do our best.

We also show our sincere appreciation to Professor William Ingram and Professor Sung Hee Park, and thank them for their helpful perspective and knowledge. Their direction and support played an important role in guiding us throughout the course of the project.

Contents

Abstract ii

List of Tables vi

List of Figures vi

1 Overview 1
- 1.1 Team-wise Functionality Distribution . . . 1
- 1.2 Defining Repository . . . 2
- 1.3 How was the Project Managed? . . . 2
- 1.4 Problems Faced and Key Takeaways . . . 2
- 1.5 Solutions Developed . . . 2
- 1.6 Follow-on Work . . . 3

2 Literature Review 3
- 2.1 Electronic Theses and Dissertations . . . 3
- 2.2 Database . . . 3
- 2.3 REST API . . . 3
- 2.4 Resource Description Framework . . . 3
- 2.5 Docker . . . 4

3 Requirements 4
- 3.1 Overall Project Requirements . . . 4
- 3.2 Collecting Requirements through Conversations with other Teams . . . 4
  - 3.2.1 Team 2 . . . 4
  - 3.2.2 Team 3 . . . 4
  - 3.2.3 Team 4 . . . 5
- 3.3 Functionality . . . 5
  - 3.3.1 Database . . . 5
  - 3.3.2 File System . . . 6
  - 3.3.3 Knowledge Graph . . . 7
- 3.4 Level of Quality . . . 7
- 3.5 User Support . . . 8
  - 3.5.1 The APIs . . . 8
  - 3.5.2 The Web Application . . . 8

4 Design 8
- 4.1 Approach . . . 8
- 4.2 Tools . . . 9
- 4.3 Methodology . . . 10
- 4.4 Workflows . . . 10
  - 4.4.1 For the Curator . . . 10
  - 4.4.2 For other Teams . . . 11
- 4.5 PostgreSQL Schema . . . 12
- 4.6 RDF Schema . . . 14

5 Implementation 16
- 5.1 Timeline . . . 16
- 5.2 Database Statistics . . . 16
  - 5.2.1 Overall Statistics . . . 16
  - 5.2.2 University-wise Statistics . . . 17
  - 5.2.3 Classification Statistics . . . 18
  - 5.2.4 Text Records Statistics . . . 19
  - 5.2.5 Topic Modeling Statistics . . . 19
- 5.3 Database Records . . . 19
  - 5.3.1 ETD Table . . . 20

5.3.2 ETD Metadata Table . . . 20
5.3.3 Object Table . . . 21
5.3.4 Object Classification Table . . . 21
5.3.5 Object Metadata Table . . . 22
5.3.6 Summarisation Table . . . 22
5.3.7 Topic Model Table . . . 23
5.3.8 Topic Set Table . . . 23
5.3.9 Class Table . . . 23
5.4 Running the Models for Teams 3 and 4 in our Pipeline . . . 23
5.5 5K ETD Subset for other Teams . . . 24

6 User Manual **25**
6.1 Front-end Interface Design and Usage . . . 25
6.1.1 Current Model . . . 27
6.1.2 Wireframes for Pages Included in Future Work . . . 28
6.2 Available ETD Metadata . . . 30

7 Developer's Manual **31**
7.1 Clone GitLab Repository . . . 31
7.2 Database . . . 31
7.2.1 Importing the CSV File . . . 31
7.2.2 Prerequisites . . . 32
7.3 Back-end Development . . . 33
7.3.1 Prerequisites . . . 33
7.3.2 Getting Started . . . 33
7.3.3 Code . . . 33
7.3.4 API Testing . . . 35
7.4 Front-end Development . . . 36
7.5 Architecture . . . 38
7.6 Hosting the Service on the Cloud Container . . . 38
7.7 API Contract . . . 38
7.7.1 Initial APIs . . . 38
7.7.2 Background on APIs . . . 39
7.8 Accessing the APIs . . . 39
7.8.1 API List . . . 39
7.9 Developing with Virtuoso . . . 41
7.9.1 Setting Up . . . 41
7.9.2 Loading Knowledge Graph into Virtuoso . . . 41
7.9.3 Querying Knowledge Graph in Virtuoso . . . 42
7.10 Future Work . . . 43

References **44**

List of Tables

1 Timeline Details . . . 16
2 Database Statistics . . . 17
3 Univerity-wise Statistics . . . 18
4 Classification Statistics . . . 18
5 Text Record Description . . . 19
6 Sample Database Record for “etd” Table . . . 20
7 Sample Database Record for “etd_metadata” Table . . . 21
8 Sample Database Record for “object” Table . . . 21
9 Sample Database Record for “object_classification” Table . . . 22
10 Sample Database Record for “object_metadata” Table . . . 22
11 Sample Database Record for “summarisation” Table . . . 22
12 Sample Database Record for “topic_model” Table . . . 23
13 Sample Database Record for “topic_set” Table . . . 23
14 Sample Database Record for “class” Table . . . 23

List of Figures

1 The current structure of the file system . . . 7
2 An RDF graph with two nodes (subject and object) and an edge connecting them (predicate) . . . 7
3 High Level Architecture . . . 9
4 Workflow of services provided for the “curator” persona. . . . 11
5 Workflow of services provided for other teams by Team 1 . . . 12
6 Updated PostgreSQL Schema . . . 13
7 Updated Knowledge Graph Schema . . . 14
8 Topic Model Table Record Example . . . 19
9 Saved Results in the file system after running the model code in Team 1 container . . . 24
10 Curator Front-end Page . . . 26
11 ETD Statistics Page . . . 27
12 Upload New ETD Page . . . 28
13 Consistency Check Page . . . 29
14 ETD Metadata Edit Page . . . 30
15 Directory Structure for ETD CSV File . . . 32
16 Sample Code for the Summarisation Model along with Schema Method Call . . . 34
17 Sample Code for Routing . . . 35
18 Import Postman Collection . . . 36
19 Front-end Modules . . . 37
20 Loading Knowledge Graph into Virtuoso . . . 42
21 Virtuoso Response . . . 42

1 Overview

Electronic Theses and Dissertations (ETDs) are digital documents that report the research and theses of graduate students. Initially, these documents were stored and archived as hard copies (printed documents), making it difficult to protect and access them. On January 1, 1997, Virginia Tech became the first university to require electronic submission of theses and dissertations. This allowed students to submit electronic documents in the form of PDFs and other attached files in various formats, such as audio, video, images, data, and many other media formats. Following that, in the past two decades, many universities around the country have also required electronic submissions. This subsequently led to a plethora of ETDs from various digital libraries at universities across the country.

The presence of these files in digital format gave access to researchers, graduate students, and anyone wanting to learn, at the click of a button from anywhere. Moreover, this allowed experimenters to create recommendations, search for relevant documents from the huge collection of ETDs, segment the documents into chapters, and build knowledge graphs to get the maximum use of these documents. Doing so will expand the accessibility and reach of graduate students' research work to anyone across the globe instead of just being archived in some libraries in some parts of the world.

In this work, we want to provide access to the ETD data and the necessary services required for the personas "Curator" and "Experimenter." For the curator, we provide a webpage that lets them view the ETD metadata, update them, and view, add, update, or delete particular ETDs and other document artifacts such as figures, tables, chapter text, etc. For the experimenters, we provide access to the database and the file system containing the ETDs, and we store their results back in the repository, by providing them with a suite of application programming interfaces (APIs). For Team 2, we provide the entire repository (metadata, full text, figures, tables, segmented chapters, and other files) to perform indexing, searching, and recommendation.

1.1 Team-wise Functionality Distribution

The course project aims to create a search engine for electronic theses and dissertations. The entire class has been divided into five teams with the following roles:

Team 1: The role of Team 1 is to assume the persona of the curator of ETDs and also that of an experimenter working with content and representations. Team 1 is responsible for storing the information by providing a repository consisting of a database, file system, front-end webpages, and knowledge graph. Finally, Team 1 is responsible for creating and maintaining a rich suite of APIs that allow the other teams to create, access, and update digital objects in the repository.

Team 2: Assuming the persona of the end user, which can be either a researcher or a graduate student working on an ETD, Team 2 is responsible for indexing the stored results in the repository offered by Team 1 and "owning" the code repository for the website available at `https://frontend.discovery.cs.vt.edu/`.

Team 3: As experimenters working on object detection and topic modeling algorithms, Team 3 generates results using the aforementioned algorithms. These results are then stored in Team 1's repository using API calls since Team 3 does not have direct access to the database and the file system.

Team 4: As experimenters working on language models, classification, and summarization algorithms, Team 4 generates results using the aforementioned algorithms. These results are then stored in Team 1's repository using API calls.

Team 5: Team 5 is the user experience and DevOps team, responsible for providing services such as Docker containers and hosting the services and functionalities extended by other teams on the Virginia Tech Computer Science Cloud, among others.

1.2 Defining Repository

Team 1 is responsible for the content representations for the collection of 500,00 ETDs. Working with a large collection of documents, and in turn being responsible for the storage of the initial ETD documents, and then eventually storing the results generated by Teams 3 and 4 (after performing topic modeling, summarization, segmentation, and classification), it is our responsibility to provide storage functionality to other teams through a combined repository consisting of a database, knowledge graph, and file system. Henceforth, we shall refer to these three entities collectively as the repository. We explore each of these three tools in more detail in the following sections of this document.

1.3 How was the Project Managed?

As the team responsible for the collection and representation of data, we thought it best to divide our roles and responsibilities, which were technical and logistic in nature. A major part of this project was working with other teams to design and implement a scalable database. The logistical responsibilities included leading meetings, liaising with other teams, taking meeting minutes (whether they were with our subject matter experts or daily stand-ups within the team), preparing presentations along with interim and final reports, as well as working with VTechWorks for submitting our work.

On the technical front, our responsibilities included creating a database and finalizing a schema for the same, setting up the file system organization, and creating a knowledge graph and a working website where we treat Teams 2, 3, and 4 as our end users. The other teams will find it increasingly useful to store the files and data they generate into the ETD repository using Team 1's back-end service, which is implemented as a Flask application.

At this point in time, we have a preliminary website and a finalized database schema. According to our plan, we have achieved almost all of the agreed-upon milestones and deliverables as outlined in our user stories. For this course project, we used Jira [6] to split the work, assign tasks, track progress, and follow agile software development practices.

1.4 Problems Faced and Key Takeaways

In this course, we were given the opportunity to build an information retrieval and analysis system from scratch. We worked in teams to build different modules and learned how they are connected to each other at different stages.

Like any typical software project, we encountered changes in user requirements. However, we ensured that we carefully listened to our stakeholders, prioritized feedback implementation, documented discussions, and communicated all our updates.

We began work with a subset of the larger collection, consisting of around 50,000 ETDs, in order to implement database indexing and API pagination, which ultimately reduced the API response time from minutes to just a few seconds.

We also had to keep in mind that we are storing long text directly in the database, and to provide necessary accommodations for the same.

1.5 Solutions Developed

We have created a Flask server to provide API services and a front-end portal for the curator. Flask is a micro web framework written in Python. It is light and simple. Considering that the curator portal will not be complicated, as would be an open, public-faced website, we chose Flask to create web pages for the front-end and for more APIs (for calling and operating upon the database).

1.6 Follow-on Work

The next steps for the project included implementing the database using PostgreSQL and creating a prototype that can easily perform the Create-Read-Update-Delete (CRUD) operations using the preliminary design of the website. This prototype would work on a subset of the ETD dataset and eventually could expand and be scaled to the larger dataset with 500K ETDs. Naturally, software projects are a constant work in progress, and the website design is also subject to change, keeping in mind ever-changing requirements. Additionally, we would leverage the file system up on the cloud service provided by Virginia Tech.

2 Literature Review

We examined various publications and research endeavors to better understand the job of our team and the work we had to do.

2.1 Electronic Theses and Dissertations

The ETD is a concept that was invented and first implemented by Virginia Tech in 1997. A research project led by the University Library at Virginia Tech has accumulated a collection of more than 500,000 ETDs from universities across the country [14]. These ETDs are the data source for the work of each of the five teams throughout the duration of this course. Each ETD has some metadata fields, which will be described in the subsequent sections. Along with the metadata, we have the PDF versions of the ETD documents.

2.2 Database

To implement the database, we decided to use MySQL or PostgreSQL. In his work *Postgres vs. mysql vs. commercial databases: It's all about what you need* [2], Conard juxtaposes the available relational database management systems such as MySQL, PostgreSQL [9], and many other commercial databases. Taking into account features such as having a wide variety of data types and the ability to create multiple master and slave databases, we chose to use the PostgreSQL relational database management system for our database needs.

2.3 REST API

In his work *Building REST APIs with Flask* [10], Kunal discusses in depth the process of creating and working with RESTful APIs using Flask, a web microframework written in Python. This work describes in detail how to connect MySQL and Flask using the SQLAlchemy API [1]. This helped us understand working with the web framework Flask and also extending it to provide API services. We were able to connect Flask to the PostgreSQL database using the SQLAlchemy API and to build the API endpoints and their actions using Flask.

2.4 Resource Description Framework

Resource Description Framework (RDF) [7] is a standard of the World Wide Web Consortium that was created as a data model for metadata. RDF graphs are multimodel directed graphs containing triples to denote the subject, the relation between the subject and object, and the object itself. We assisted Team 2 by providing them with access to the Virtuoso [13] database engine that allows them to store RDF data in a relational database [5].

2.5 Docker

Ian et al. in their work *Docker In Practice* [8] discuss in depth about creating and working with Docker containers [3] and the advantages of using Docker images to set up an efficient and clean CI/CD pipeline. This work familiarized us with the concept of containers and helped us understand their advantages and usage.

3 Requirements

This section will discuss what our team needed to accomplish in terms of functionality, level of quality, and user support.

3.1 Overall Project Requirements

During the initial phase of the project, our team sought to understand the existing data set, identify the key deliverables to be achieved at the end of the semester, as well as initially identify and establish a database schema by collaborating with other teams and identifying the kind of data that they will generate.

On top of this, we wanted to provide necessary services, such as a webpage that would provide a view and modify functionality, when, as a team, we assumed the persona of a curator.

Moreover, as experimenters responsible for content and representations, we wanted to be able to allow other teams to access the database and the file system.

3.2 Collecting Requirements through Conversations with other Teams

3.2.1 Team 2

To better understand whether our database schema and API designs met the user requirements, we had further discussions with each team about our initial design proposal. Team 2 needs to read all the data and index it using Elasticsearch [4]. They do not require any write APIs nor to add data to the database. One of their major requirements is to retrieve all of the data in the database and synchronize it with their system every day. However, because of the huge volume, an API that returns all database records is inappropriate. We proposed a contract that returns a paginated response. The nested JSON response contains a list of ETD records with the determined objects of each ETD. The API can be called in batches to index all data in the database. The other requirement for Team 2 is an RDF triplestore to store relations about the digital objects. We have implemented that using Virtuoso.

In conclusion, Team 2 has two main requirements:

- API for retrieving all ETD collections and related data.
- Set up RDF and publish an API to read data.

For the first requirement, we finished several APIs for Team 2, which can be found in Section 7.8.1. For the second requirement, we have set up the knowledge graph with sample data; see Section 7.9. The created API can be also found in the API list in Section 7.8.1.

3.2.2 Team 3

Team 3 plays an important role in the determination of objects. They generate an XML document as part of their object detection work. The XML document consists of all the objects detected for an ETD, and related metadata.

Team 3 initially just wanted to save the entire XML without further processing. One of the reasons that they wanted to save XML was that they would reuse it for the front-end page view, and felt it was unnecessary to translate it to JSON and change it back again. Based on Dr. Fox's suggestions, both teams mutually agreed on a JSON API contract. We also provided a solution so that we could store the entire XML for quick access.

They also work on topic modeling. We proposed an API contract that takes information about the digital object, the topics associated with it, and related digital objects. For storing object detection results, we decided to create an API that adds the details of a single object to the database. The API could be used by Team 3 whenever they detect a new object.

We also need to provide a batch upload API that accepts a ZIP file and a list of objects in JSON.

In conclusion, the two requirements of Team 3 are:

- API to save the Detected Object results along with XML file.
- API to save the Topic Modeling results.

For the first requirement, we created the "Save Detected Object" API for them to store every object Team 3 detected, along with an XML file. For the second requirement, we created the "Save Topic Modeling Results" API. These can be found in Section 7.8.1.

3.2.3 Team 4

Team 4 works on segmentation, classification, and summarization. Similar to Team 3, we have to save all the results. The difference between Team 4's segmentation and Team 3's object determination is: Team 4's segmented chapter includes figures, but Team 3's chapter object is pure text. Team 3 asked us to store the segmentation information separately in different tables to distinguish between them. Then Team 4 agreed with the solution we provided: we treat their segmented chapter as an object and store it in the object table shared with Team 3, but we would give it a special type ("chapter-se") to distinguish their chapter and Team 3's chapter. Currently, the algorithm Team 4 uses for classification does not provide more data other than the output class labels, but they informed us that different models might require and produce different attributes. However, at the end, there was no conflict in the naming convention for the digital objects between Teams 3 and 4 as Team 3 does not produce objects with the type "chapter". All "chapter" type objects are produced by Team 4. So "chapter-se" is no longer needed.

In conclusion, Team 4's two requirements are:

- API to receive chapter objects in order to perform summarization and classification.
- API to save summarization and classification results.

For the first requirement, Team 3 can also use the "Save Detected Object" API to store a chapter as a type of object. For the second requirement, we created the "Save Summarization" and "Save Classification" APIs. These can be found in Section 7.8.1.

3.3 Functionality

Being responsible for ETD curation, as well as the representation of content and metadata throughout the duration of this semester, Team 1 strives to provide functionality in three flavors. Each is discussed below.

3.3.1 Database

As a team responsible for content and representations, one of our primary duties is to store the ETD data in an effective manner using a repository. For this purpose, we collaborated with Teams 3 and 4. We understood their

requirements such as the kind of data they will be generating. For Team 3, it is the output of their object detection and topic modelling. In the case of Team 4, it is from language modelling, segmentation, classification, and summarization.

After this exercise, we identified a general outline for the schema of our database. We would be implementing a relational database system containing various attributes in order to store the data in said tables. Another key objective is to make this database, backed by the file system, highly scalable.

3.3.2 File System

The file system is in the form of a tree-like directory structure. On a preliminary basis, the ID that can be mapped to an ETD consists of digits — the first few of which correspond to the parent folder that the ETD belongs to, and the remaining digits pertain to the ETD number. The file system aims to correctly store and structure these ETD files and to finally be able to store instances of chapters, summaries, tables, figures, etc., that will be provided by other teams.

The file system is located in the "etdrepo" folder on the Camelot server. This could directly be mounted on the container and is provided with read access. We copied a subset of ETDs with the current file structure and populated the PostgreSQL database.

In order to understand the file system, we can start from the top of the tree diagram. The main "etdrepo" contains over 50 folders with three-digit numbers assigned to each. Currently, they represent the first level of the file system. Each three-digit folder contains around 5000 to 10,000 lower level folders numbered from 0000. Each lower level folder is associated with an ETD and contains digital objects like PDF, cleaned text, parsed text, and the XML file. The "*.pdf" files are the ETDs' PDF versions. "*.xml" files are generated by Team 3 as the output to their object detection pipeline."*.html" files are generated by parsing the XML from Team 3. "*_Parsed.txt" files are older clean text files (files that were already present in the 500k file system and were not generated or worked on by any of the teams for this project) and are available for certain ETDs. However, they are not used as a part of this project. "*_pdf_cleanText.txt" files are generated by Team 4 as extracted text files from PDF versions of ETDs to perform summarization on. We create a separate directory for each classified object as well. Hence, after detection by Team 3, the extracted objects like images, equations, etc. can be stored in sub-folders of the corresponding ETD digital objects.

We do not use the 500k dataset for this project. For the purpose of this project, we are using a subset of ETDs. From the 500k ETDs we sampled ETDs of 5 universities (amounting to 57,129 ETDs) and created a similar file system. To support access to the file system, we have coordinated with Team 5 to mount this file system for each team's container. We provide APIs to other teams for the purpose of injecting and fetching data from the file system as well as the database according to the requirements. The current structure of the file system is reported in Figure 1.

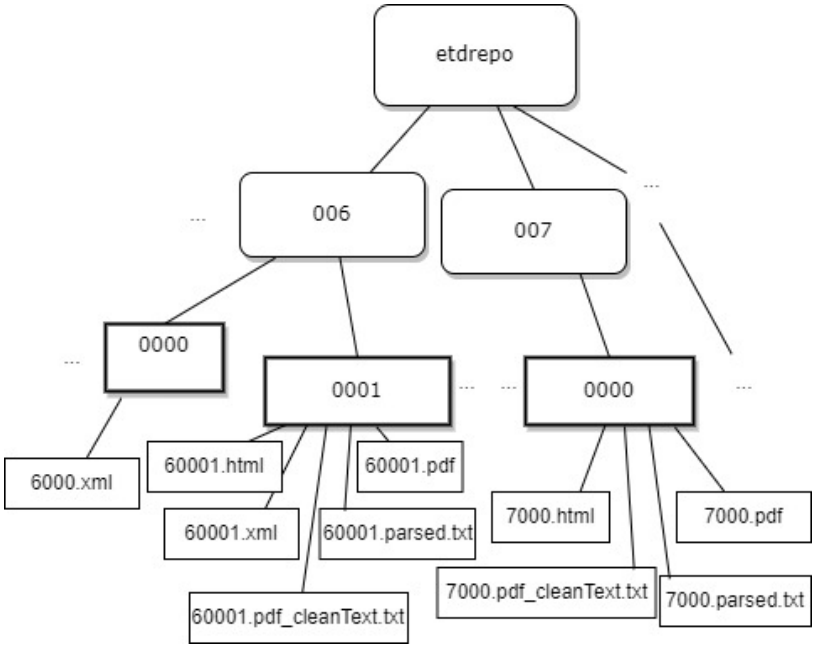

Figure 1: The current structure of the file system

3.3.3 Knowledge Graph

The Knowledge Graph aims to represent the database entities in a network-like structure in order to make them more accessible through queries. With knowledge graphs, one can easily explore and map entities and their relationships with each other.

The Resource Description Framework (RDF) is a framework for representing information using a knowledge graph. The core structure of the abstract syntax is a set of triples, each consisting of a subject, a predicate, and an object. A set of such triples is called an RDF graph. An RDF graph can be visualized as a node and directed-arc diagram, in which each triple is represented as a node-arc-node link, as shown in Figure 2. For our use case, we would store the digital objects extracted by Teams 3 and 4 at the nodes and the relationship between them as the predicate, as reported in Figure 7.

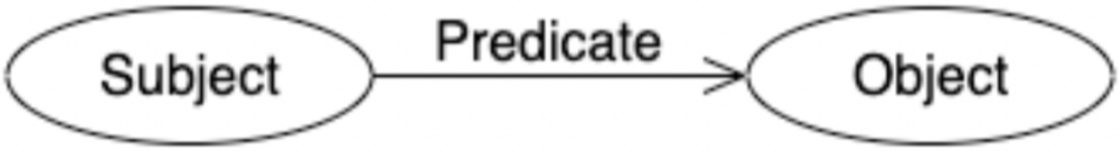

Figure 2: An RDF graph with two nodes (subject and object) and an edge connecting them (predicate)

3.4 Level of Quality

Team 1 strives to deliver quality work in terms of creating a scalable architecture that satisfies all the requirements of the curator. The quality of the work with respect to the goals would be measured with respect to the following rubric.

1. Strive for curatorial satisfaction

- All the services that the curator might need should be easily accessible to them via the APIs.
- The upload API should be able to support all types of digital objects.

2. Database support for other teams
   - The database should be functional, with a schema that supports all the data needs of other teams.
   - The item database should be scalable and support streams of API calls.
3. Item file systems should be scalable and be able to structurally support digital object needs.
4. All services required by the experimenters should be available.
   - We want to deliver an interactive website with options to modify and view content in an attractive format.

3.5 User Support

The Curator, who is the user for this project, needs to perform operations to create, retrieve, update, and delete ETDs. For ease of use, a CRUD application with an interactive front-end has been developed to support user operations. The proposed application uses a micro web framework, that is, Flask, to generate a dynamic website.

3.5.1 The APIs

We also provide application programming interface (API) support. The APIs we would provide will help users communicate with the database using a set of definitions and protocols. APIs would be used to retrieve ETD information and inject new data into the database.

We have a huge dataset, which impacts the latency of the API. Some users need support in terms of bulk database operations. The larger dataset poses a challenge because API operations become tedious when sorting through the huge amount of data. In this case, we provide support through REST APIs to reduce latency and provide support for handling streams of data.

3.5.2 The Web Application

This section showcases the layout of the webpage for the curator. The front-end of the webpage is implemented using HTML/CSS and JavaScript. The front-end is implemented in a manner that makes it easier to perform the previously mentioned CRUD database operations.

4 Design

4.1 Approach

From the high-level overview of the system shown in Figure 3, we have five major components in the system design: the server, the PostgreSQL and RDF databases, the file system, the Flask web application for the curator, and the APIs for the Experimenters. The server (cloud.cs.vt.edu) has the DLRL cluster where we have our Docker container with PostgreSQL, Flask, and other necessary packages. The database is present in this Docker container. The file system is mounted from the Docker container, and the database has information about the location of each ETD in the file system. We have a separate Docker container for the RDF database, which can be accessed using the APIs that we have designed. The Flask application runs on another container on the server, which not only hosts the web application that the Curator uses, but also provides the APIs that the other teams would need to interact with the database and the file system.

We worked with four other teams, with whom we interacted at different stages. Our team prepared the file system in such a way that read and write performance is not affected when other teams are reading or inserting data like object images, chapter summaries, etc. We have built a web portal in Flask where curators can access the ETDs using the Flask framework in Python. It has an API to do CRUD operations. Team 5 helped us with setting up the K8 Cluster and Docker containers. The ETDs are present on a server different from our database server, and we have used the local path of the files on that server in our database.

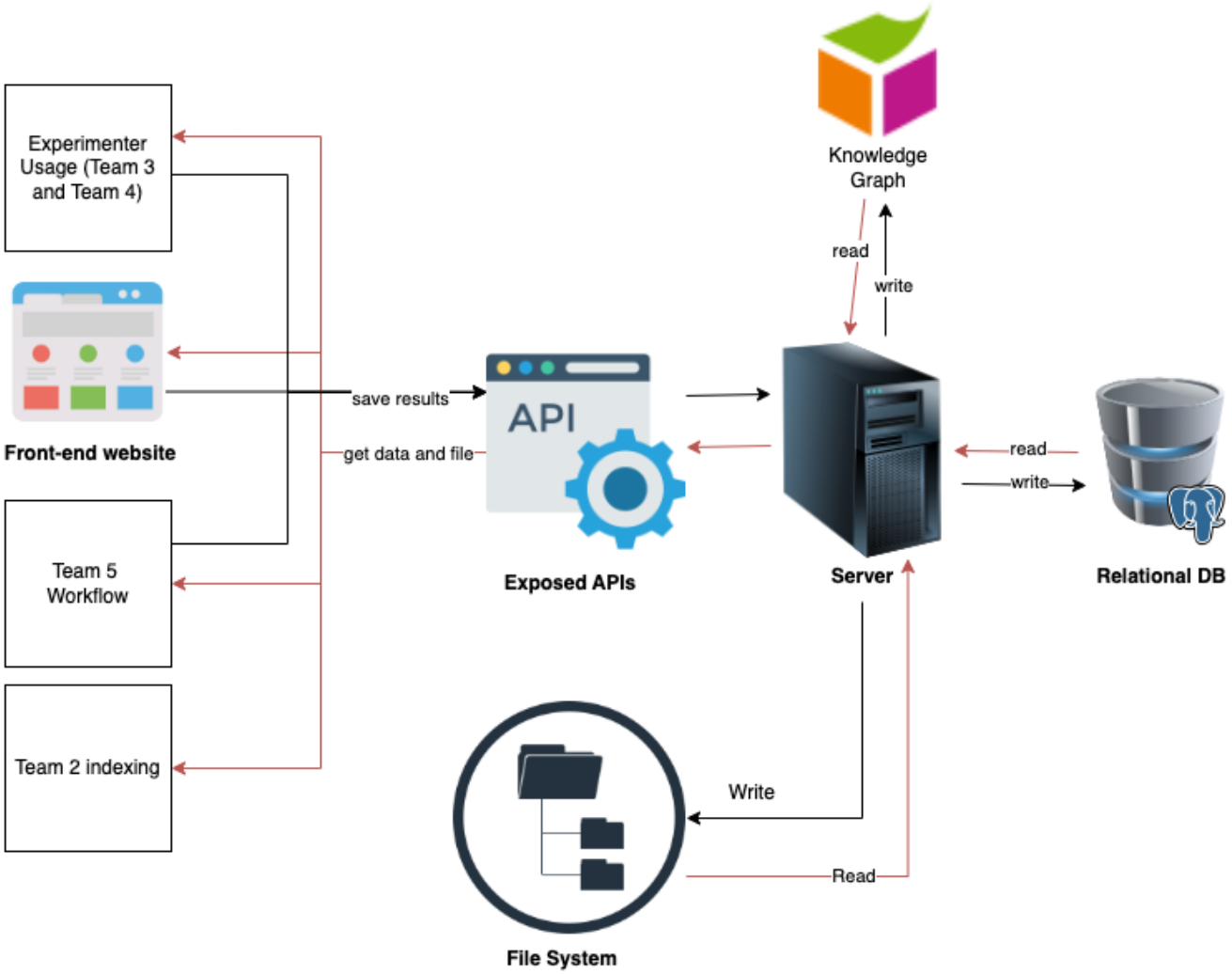

Figure 3: High Level Architecture

4.2 Tools

- Almost all of the source code for our project is implemented in Python. Each service is written in Python, using both standard and external libraries (such as `gensim`).
- Docker is used to containerize our services. It is intended to modularize our work and provide an easy interface for future work to build off of.
- Flask is used to create the webpage for the curator and also the APIs for the experimenters to use.
- PostgreSQL is used as the database to store ETD metadata and information about chapters and figures. Each row has a column that points to a location in the file system where the file actually exists.
- Virtuoso is used to store the RDF triples that Team 2 would need for searching and indexing. This database can store the relations between the objects that Team 3 provides.

- Postman is an API platform that allows developers to create, test, and iterate their APIs. We used Postman to design and document API contracts required by the other teams.
- Vertabelo is a database modeler that allows you to create visual databases online. We used Vertabelo to design the initial database schema and also to keep it updated with the changing requirements.

4.3 Methodology

The work of Team 1 is associated with handling the operations and metadata of over 500,000 Electronic Theses and Dissertations (ETDs). This data has been collected over the years from academic institutions all over the country. Of these 500,000 ETDs, almost all local ETD data is obtained from VTechWorks, the official repository of electronic thesis and dissertations for Virginia Tech. For the duration of the semester, we assumed two personas, that of a curator of ETDs and the associated metadata, and that of an experimenter working with content and representations. With the help of our SMEs, we were able to define our methodology in terms of user stories.

- User Story 1: As a curator, I want to add a new ETD to the file system and database.
- User Story 2: As a curator, I want to update the data for an existing ETD.
- User Story 3: As a curator, I want to check different statistical reports of the current collections.
- User Story 4: As a developer in Team 3, I want to store the output of the object detection in the database and file system.
- User Story 5: As a developer in Team 4, I want to get the necessary and well-organized data for me to create segmented chapters and then summarize and classify them and eventually store them in the database and file system.
- User Story 6: As a developer in Team 2, I want to index all data in the database and file system to set up ElasticSearch.

4.4 Workflows

4.4.1 For the Curator

For the persona of "Curator", the following services are provided by Team 1:

- Add a single or a stream of ETDs
- Consistency check of the file system and the database
- Statistical reports for the collection of ETDs
- Ability to edit ETD metadata

The flow chart displayed in Figure 4 explains how these services are provided.

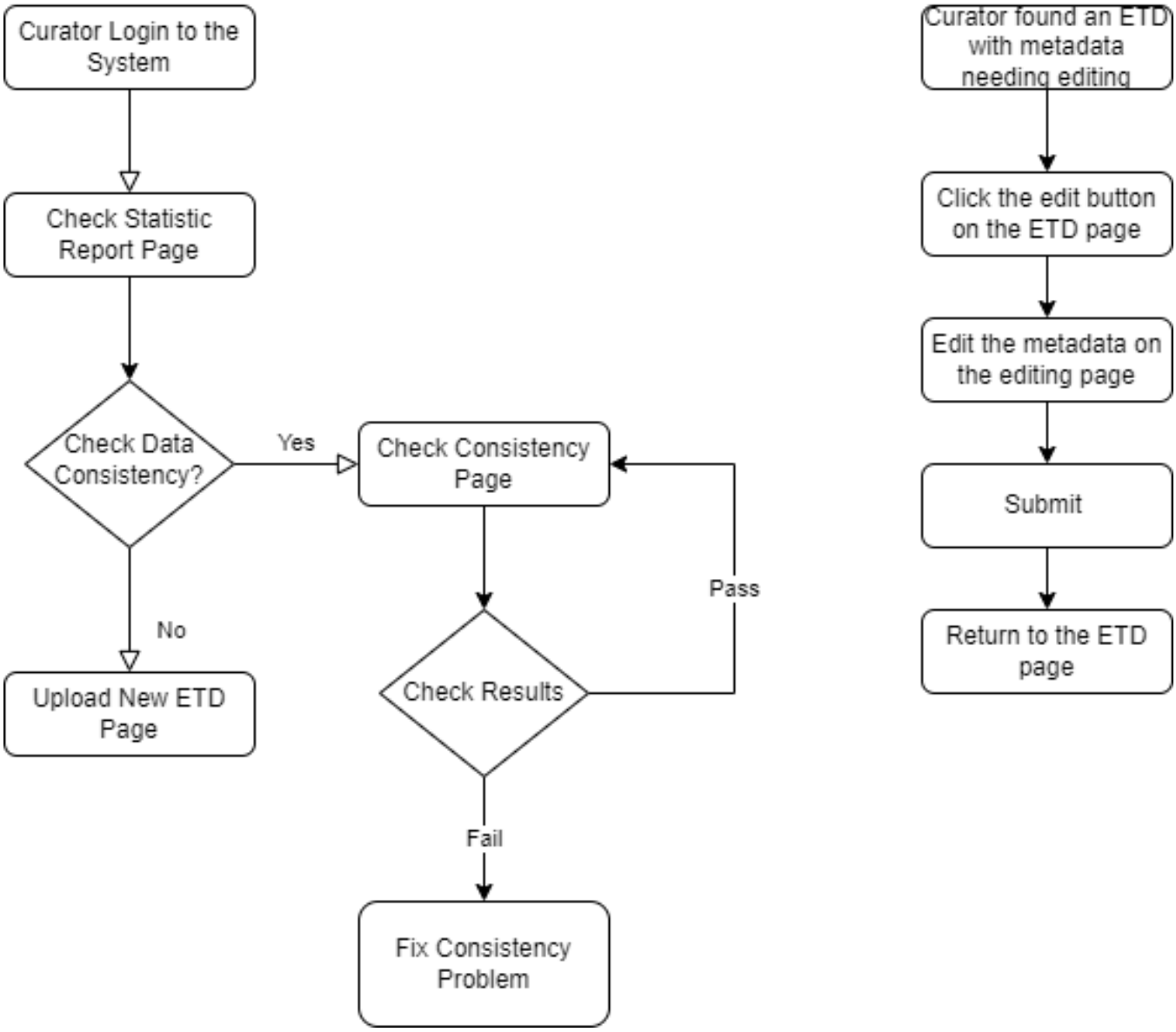

Figure 4: Workflow of services provided for the "curator" persona.

4.4.2 For other Teams

As Team 1, we provide data access to other teams to different services, which is a way to interact with the repository. Figure 5 shows the workflow of the services we provide for each of Teams 2, 3, and 4.

Here, "Indexing for ElasticSearch" is meant for Team 2, "Object" functionality is meant for Team 3, and "Summarization" and "Classification" functionalities are intended for Team 4.

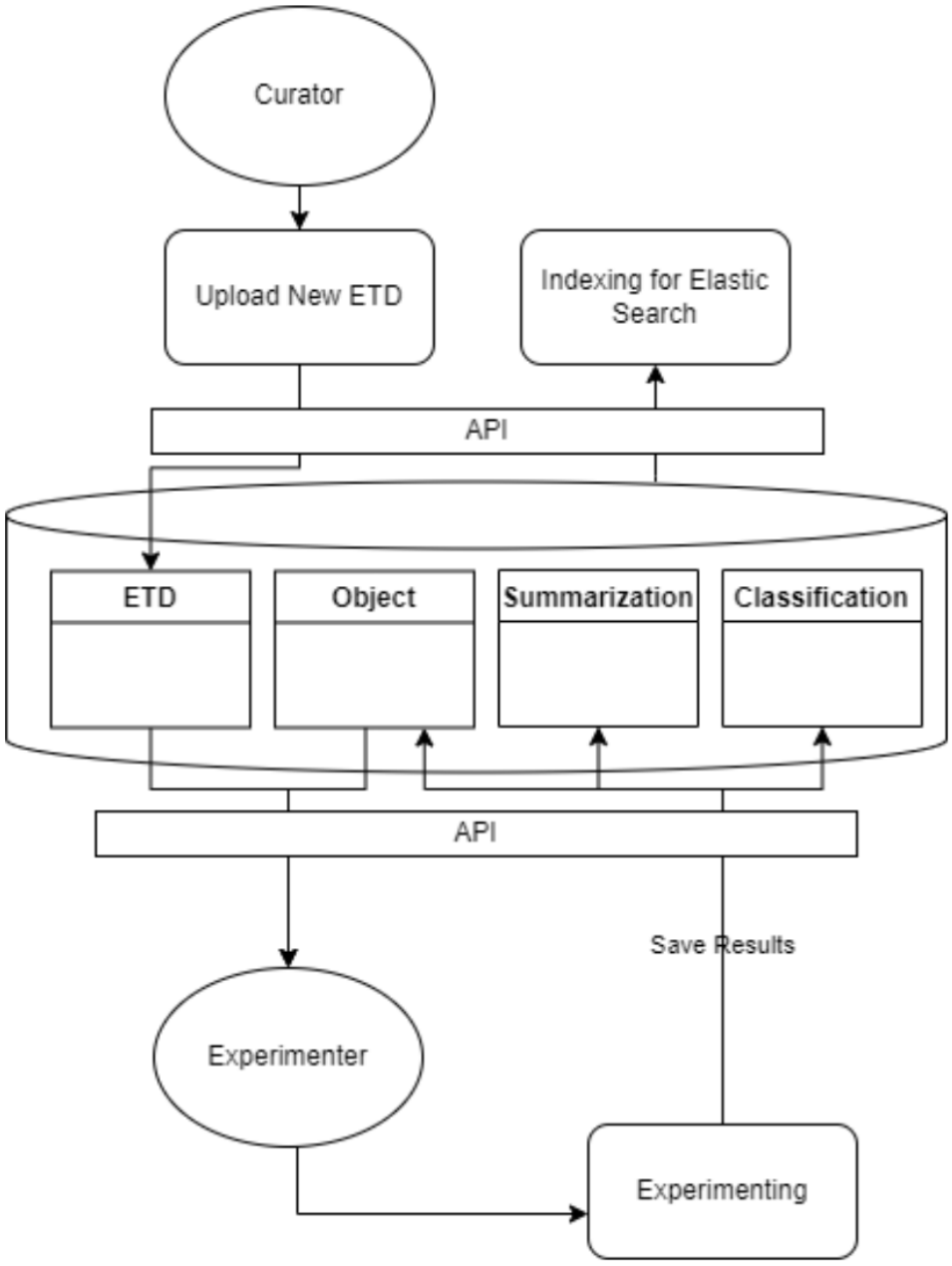

Figure 5: Workflow of services provided for other teams by Team 1

4.5 PostgreSQL Schema

After getting the suggestions from Dr. Fox in Interim Report 1, discussing with our SMEs, and clarifying the requirements from other teams, the design was changed and started to be roughly fixed. The final database schema can be seen in Figure 6 below. This schema has been ratified with Dr. Fox, our SMEs – Professors Ingram and Park, as well as the other teams. The following class diagram was created using the Vertabelo [12] database modeler.

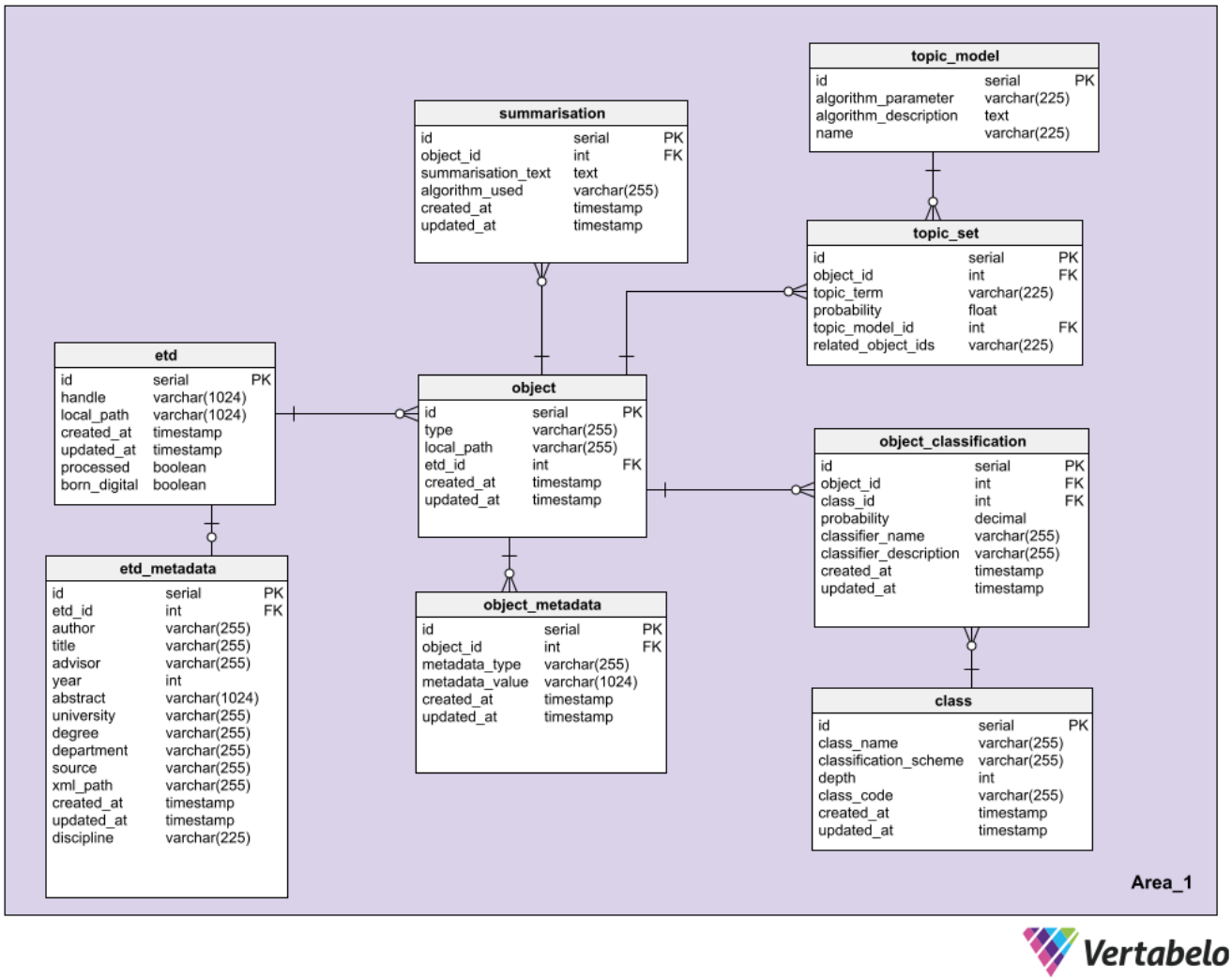

Figure 6: Updated PostgreSQL Schema

When the user adds a new ETD to the system, we first register it in the "etd" table. The "etd" table records the path for each ETD that allows us to find the file in the file system. When object detection has been performed, the resultant objects and metadata will be updated into the tables "etd_metadata", "object" and "object_metadata". The "object_metadata" table was designed for scalability. We abstracted the metadata attribute name and value to "metadata_type" and "metadata_value". This allows the table to accept different possibilities for each object type.

Our SMEs suggested that we can also abstract the attributes in the "etd_metadata" table. Based on our current progress, we thought it best to leave the design as is for now, as it matches the CSV files that we received as part of the previous work that has been conducted on this topic. Additionally, our progress in terms of code development is in coherence with the implementation of the corresponding models.

The topic modeling, summarization, and classification processes have the objects as their experimental subjects. These then store the results in the corresponding table. We also provided space to record the algorithm, model, and scheme that are used for the experiment in our PostgreSQL database schema.

4.6 RDF Schema

We have stored the information about digital objects and their corresponding relationship in the RDF triplets as shown in Figure 7. The `object_id` has the value of the identifier field of the object table in the relational database. The path contains the value of the local path of the digital object in the file system. We also store `etd_id` to identify the ETD with which the digital object is associated. While creating the knowledge graph we also initialized some relationship objects having common values like `has_a_figure` and `has_a_chapter`, etc.

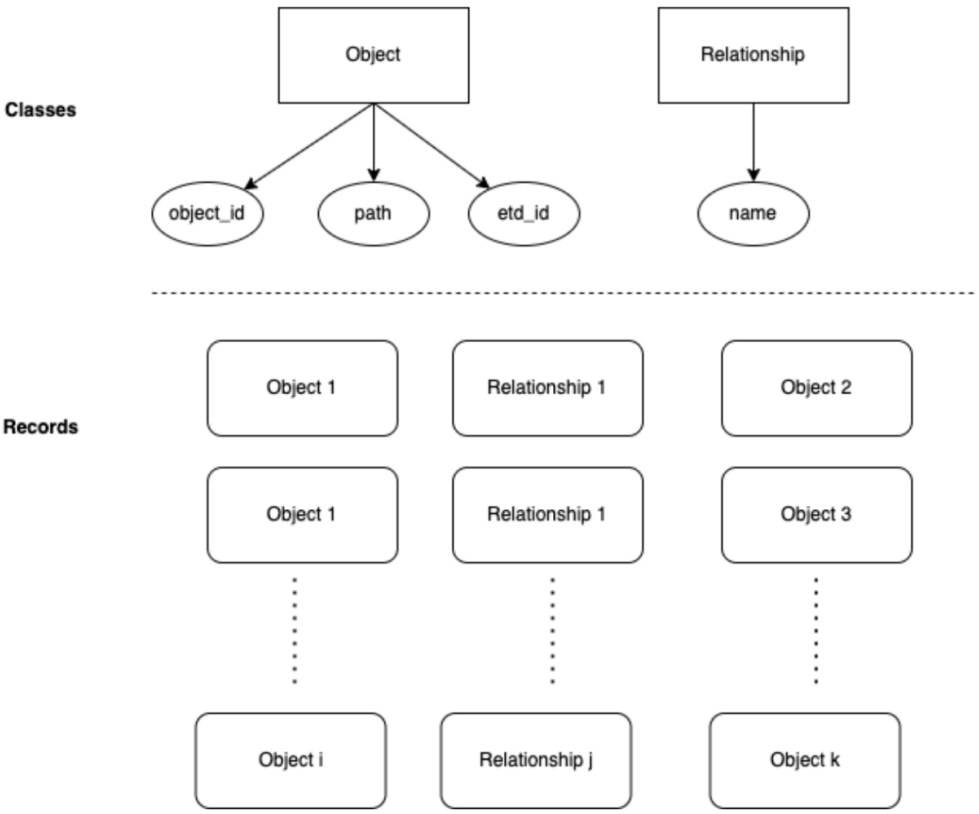

Figure 7: Updated Knowledge Graph Schema

We have loaded the data model written using Turtle (Terse RDF Triple Language [11]) format (see below) into Virtuoso. The Turtle format is a textual syntax for RDF that allows an RDF graph to be completely written in a compact and natural text form, with abbreviations for common usage patterns and data types.

```
@prefix dbo: "http://dbpedia.org/ontology/".
@prefix dbp: "http://dbpedia.org/property/".
@prefix dbpedia: http://dbpedia.org/resource/ >.
@prefix rdf: "http://www.w3.org/1999/02/22-rdf-syntax-ns#".
@prefix rdfs: http://www.w3.org/2000/01/rdf-schema#>.
@prefix to: http://example.org/tuto/ontology#>.
@prefix ttr: http://example.org/tuto/resource#>.
@prefix xsd: "http://www.w3.org/2001/XMLSchema#".
```

```
tto:Object a rdfs:Class ;
rdfs:isDefinedBy <http://example.org/tuto/ontology#> ;
rdfs:label "object" .

tto:Relationship a rdfs:Class ;
rdfs:isDefinedBy <http://example.org/tuto/ontology#> ;
rdfs:label "relationship" .

tto:object_id a rdf:Property ;
rdfs:domain tto:Object ;
rdfs:isDefinedBy <http://example.org/tuto/ontology#> ;
rdfs:label "object_id" ;
rdfs:range xsd:integer .

tto:etd_id a rdf:Property ;
rdfs:domain tto:Object ;
rdfs:isDefinedBy <http://example.org/tuto/ontology#> ;
rdfs:label "etd_id" ;
rdfs:range xsd:integer .

tto:path a rdf:Property ;
rdfs:domain tto:Object ;
rdfs:isDefinedBy <http://example.org/tuto/ontology#> ;
rdfs:label "path" ;
rdfs:range xsd:string .

tto:name a rdf:Property ;
rdfs:domain tto:Relationship ;
rdfs:isDefinedBy <http://example.org/tuto/ontology#> ;
rdfs:label "name" ;
rdfs:range xsd:string .
```

5 Implementation

5.1 Timeline

Table 1 shows the set of tasks and deliverables along with their dates of completion.

Task	Date
Define Project Roles	08/25/2022
Review Key Literature	08/30/2022
Talk with SMEs about Goals and Requirements	08/31/2022
Collaborate with other Teams to Collect Requirements	09/06/2022
Create a Simple Flask Application	09/08/2022
Interim Report 1 Presentation	09/13/2022
Propose a Database Schema	09/14/2022
Interim Report 1	09/15/2022
Update the API Contracts	09/26/2022
Create Initial API Prototypes	10/02/2022
Finalize the Database Schema	10/05/2022
Move the Service to the Docker Container	10/05/2022
Create the Initial Knowledge Graph on Local Environment	10/08/2022
Interim Report 2 Presentation	10/04/2022
Interim Report 2	10/11/2022
Create Read API for Team 2 to Access Knowledge Graph	10/02/2022
Update the Database Schema	10/13/2022
Copy Files to the File System	10/14/2022
Populate the Database	10/15/2022
Update APIs for Team 2	10/18/2022
Load ETD Records in the Knowledge Graph for Schema Verification	10/27/2022
Discuss with Team 3 about the Requirements for the Embedded Code	10/28/2022
Discuss with Team 4 about the Requirements for the Embedded Code	10/31/2022
Check the Container's Abilities to Write to the File System	11/01/2022
Interim Report 3	11/03/2022
Implementation for Team 3 - Save Page Images	11/13/2022
Implementation for Team 3 - Save Object Determination Results	11/14/2022
Implementation for Team 3 - Save Topic Modelling Results	11/16/2022
Implementation for Team 4 - Save Chapter Results	11/17/2022
Build Flow to Insert Data into Knowledge Graph when New Data is Inserted into PostgreSQL	11/17/2022
Embed Code from Teams 3 and 4	11/17/2022
Create Web Pages for the Curator	11/29/2022
Final Report	12/12/2022

Table 1: Timeline Details

5.2 Database Statistics

5.2.1 Overall Statistics

Although we initially had a data set comprising 500,000 ETDs, we decided to work on a subset of ETDs for this course. Our aim was to build a prototype of a smaller number of ETDs working first, and then, eventually, one key point in the future works might include scaling this system to the complete set of ETDs.

Some of the key statistics for the database are described in Table 2.

Entity	Number
ETD Records	57,129
Object Records	1,301,815
Born-digital ETD Records	32,080
Summarization Records	22,617
Chapter Records	22,845
"Cleaned Text" Records	22,849
Image Records	166,489
Page Records	225,531
Text Records	858,086
XML Records	1,472

Table 2: Database Statistics

Table 2 encapsulated the database metrics. We started with 57,129 ETD records spread across six universities - University of California, San Francisco; Georgia Insitute of Technology; University of California, Berkeley; California Institute of Technology; University of California, Los Angeles; and the North Texas State University.

After Teams 3 and 4 applied various deep learning models and generated results based on entire ETD documents, using topic modeling, summarisation, segmentation, and classification algorithms, we were presented with a host of "objects" which included chapter summaries in the form of text, classification labels which are variable-length strings, generated topic sets which consisted of topic terms, probabilities, topic IDs, et cetera. The outcome of these generated results was a total of 1,301,815 which is basically more than 1 million in number. The reader might refer to Figure 6 in order to analyse the attributes that are spread across the relational database system spread across nine tables.

One interesting thing to note about the subsets of ETDs that we were presented with in the beginning of the semester is the distinction between "born-digital" and "scanned documents". Born-digital documents are those that were electronically-typed ever since their respective inceptions. On the other hand, scanned documents are those which were first typed and then printed and scanned; these tend to be older. In our database, we have 32,080 born-digital documents.

The summarisation records are 22,617 in number and consist of chapter summaries. The chapter records are 22,845 in number and comprise of the segmented chapter PDF documents that are saved in the file system. "Cleaned Text" records are text-parsed documents generated from the segmented chapter PDFs and are 22,849 in number. All of these results have been due to Team 4's algorithms, run on a subset of the ETDs numbering 5,000, as is explained in Section 5.5.

The 166,489 image records are provided by Team 3 through their object detection algorithms; these include figures, equations, tables, and any algorithm descriptions within the ETDs. Similarly, Team 3's algorithms also produced 225,531 records which are basically page images. Lastly, the 858,086 text records are also provided by Team 3 and are discussed in depth in Section 5.2.4. It should be noted that all of these counts of records produced by Team 3 came from a subset of the subset of the ETDs numbering 5,000 (explained in Section 5.5). At the time these counts were made, Team 3 had not completed object detection for all of the 5,000, else the numbers given earlier in this paragraph would all have been proportionally larger.

5.2.2 University-wise Statistics

From Tables 2 and 3, the reader can infer that there are a total of 57,129 ETD records available in the database. These 57,129 documents belong to six universities, as can be seen in Table 3. We decided to randomly pick six universities and store their records within the database; no specific selection process was involved.

University	Number
University of California, San Francisco	2,186
Georgia Insitute of Technology	22,399
University of California, Berkeley	8,252
California Institute of Technology	10,363
University of California, Los Angeles	8,107
North Texas State University	5,822

Table 3: Univerity-wise Statistics

However, it is to be noted that within the database, abbreviations were used for University of California, San Francisco; University of California, Berkeley; and University of California, Los Angeles – "ucsf", "ucb", and "ucla", respectively – with no changes to the other three aforementioned universities.

5.2.3 Classification Statistics

Classification labels are produced by Team 4's efforts. The statistics concerning them essentially classify ETDs in what we call "subjects" or "disciplines".

We have a total of 27 classes. These were generated by the "SciBERT" model. Table 4 shows the number of records for each class.

Class Name	Number
City Planning	860
Applied Physiology	348
Civil and Environmental Engineering	2,584
Public Policy	3,855
Chemical Engineering	1,037
History, Technology and Society	1,400
Engineering	2,035
Architecture	1,699
Management	990
Chemistry and Biochemistry	2,851
Economics	723
Psychology	579
Biomedical Engineering	1,922
Literature, Communication and Culture	3,927
Computer Science	2,908
Building Construction	348
International Affairs	3,543
Aerospace Engineering	593
Industrial and Systems Engineering	1,845
Earth and Atmospheric Sciences	1,176
Mechanical Engineering	1,520
Biology	1,807
Electrical and Computer Engineering	2,806
Mathematics	738
Materials Science and Engineering	1,276
Polymer, Textile and Engineering	214
Physics	2,278

Table 4: Classification Statistics

5.2.4 Text Records Statistics

This section is essentially a continuation of Section 5.2.1, which gives an overview of the database statistics. The text records are generated by the object detection algorithms designed by Team 3.

The text records in the database are 858,086 in number and are of a host of types which are enumerated in Table 5:

Text Attribute Name	Description
title	ETD Title
abs_heading	Abstract Heading
abs_text	Abstract Text
author	Name(s) of the Author(s)
committee	Name of Committee
date	Date of Publish of ETD
degree	Associated Degree to be Awarded
equation number	Equation Number
figure caption	Figure Caption
footnote	Footnote Text
name	Student's Name
page number	Page Number
paragraph	Paragraph Text
ref_heading	References' Heading
ref_text	Text in References
table caption	Table Caption
title	Heading Title
toc_heading	Table of Contents Heading
toc_text	Table of Contents Text
university	Name of the University

Table 5: Text Record Description

5.2.5 Topic Modeling Statistics

For the topic model results, we saved the topic set in a string array, so we were not able to derive meaningful statistics for those results. This can be a part of future work.

Figure 8 shows an example of a record in the topic modeling table. It indicates what topic terms are related to the object, and what object is related or similar to the object based on the similarity of the topic that has been detected. The topic model used is "CTM_25" designed by Team 3.

id	object_id	topic_term	probability	topic_model_id	related_object_ids
94	196,119	{galaxy,invariant,problem,ritual,material}	(NULL)	10	{667,1246,595,1579,1075}

Figure 8: Topic Model Table Record Example

5.3 Database Records

Following the database (PostgreSQL) schema in Section 4.5, we present to the reader some actual database records. These are tabulated below and classified by the name of the database table that the data belongs to.

There are essentially nine tables in our database, namely, “etd”, “etd_metadata”, “object”, “object_classification”, “object_metadata”, “summarisation”, “topic_model”, “topic_set” and lastly, “class”. Tables 6-14 specify each of these.

Here, in order to show a sample of data in the database, we picked a random ETD whose title is: “Portrait of the Rugged Individualist: The Nonverbal Pride Display Communicates Support for Meritocracy”.

One can map the various attributes’ names within a table from Figure 6 and verify them in the tables that follow. The results shown here are a cumulative effort of all the teams in that the stored results are the results generated with the help of algorithms designed by them. As Team 1, our job was to correctly group or classify the results generated by other teams under these nine tables and to store them in the database.

Lastly, along with mentioning the records stored in each table, we also mention the Primary and Foreign Keys, wherever applicable, to tie the nine tables together.

5.3.1 ETD Table

Table 6 shows the ETD table from the database schema along with a sample record. Here, the Primary Key is “id”.

Attribute	Data Type	Value
“id”	serial	4
“handle”	varchar (1024)	“1438”
“local_path”	varchar (1024)	“0001438”
“created_at”	timestamp	“2022-10-20T18:39:04.195Z”
“updated_path”	timestamp	“2022-10-20T18:39:04.195Z”
“processed”	boolean	“false”
“born_digital”	boolean	“true”

Table 6: Sample Database Record for “etd” Table

5.3.2 ETD Metadata Table

Table 7 shows the ETD Metadata table from the database schema along with a sample record. Here, the Primary Key is “id” and the Foreign Key is “etd_id”.

Attribute	Data Type	Value
“id”	serial	4
“etd_id”	int	4
“author”	varchar (255)	“Horberg, Elizabeth Jane”
“title”	varchar (255)	“Portrait of the Rugged Individualist: The Nonverbal Pride Display Communicates Support for Meritocracy”
“advisor”	varchar (255)	“Keltner, Dacher;”
“year”	int	2010
“abstract”	varchar (1024)	“Emotions profoundly influence beliefs about morality and justice (Haidt, 2001) ..... Moreover, consistent with Studies 1 and 2, observers rated the high-pride expressers as more likely to support meritocracy and less likely to support egalitarianism.”
“university”	varchar (255)	“ucb”
“degree”	varchar (255)	null
“department”	varchar (255)	null
“source”	varchar (255)	“`https://escholarship.org/content/qt0v37d9g2/qt0v37d9g2.pdf?t=mtfrpv`”
“xml_path”	varchar (255)	null
“created_at”	timestamp	“2022-10-20T18:39:04.195Z”
“updated_path”	timestamp	“2022-10-20T18:39:04.195Z”
“discipline”	varchar (225)	null

Table 7: Sample Database Record for “etd_metadata” Table

5.3.3 Object Table

Table 8 shows the Object table from the database schema along with a sample record. Here, the Primary Key is “id” and the Foreign Key is ”etd_id”.

Attribute	Data Type	Value
“id”	serial	4
“type”	varchar (255)	“chapter”
“local_path”	varchar (255)	“`000/1438/Team4_chapters/chap_1.pdf`”
“etd_id”	int	4
“created_at”	timestamp	“2022-11-16T22:46:15.540Z”
“updated_path”	timestamp	“2022-11-16T22:46:15.540Z”

Table 8: Sample Database Record for “object” Table

5.3.4 Object Classification Table

Table 9 shows the Object classification table from the database schema along with a sample record. Here, the Primary Key is “id” and there are two Foreign Keys – “object_id” and “class_id”.

Attribute	Data Type	Value
“id”	serial	62
“object_id”	int	196116
“class_id”	int (255)	24
“probability”	decimal	null
“classifier_name”	varchar (255)	null
“classifier_description”	varchar (255)	null
“created_at”	timestamp	“2022-11-27T03:33:26.940Z”
“updated_path”	timestamp	“2022-11-27T03:33:26.940Z”

Table 9: Sample Database Record for “object_classification” Table

5.3.5 Object Metadata Table

Table 10 shows the Object Metadata table from the database schema along with a sample record. Here, the Primary Key is “id” and the Foreign Key is “object_id”.

Attribute	Data Type	Value
“id”	serial	205931
“object_id”	int	286931
“metadata_type”	varchar (255)	“bbox”
“metadata_value”	varchar (1024)	“[1461, 121, 1546, 180]”
“created_at”	timestamp	“2022-11-29T05:36:14.705Z”
“updated_path”	timestamp	“2022-11-29T05:36:14.705Z”

Table 10: Sample Database Record for “object_metadata” Table

5.3.6 Summarisation Table

Table 11 shows the Summarisation table from the database schema along with a sample record. Here, the Primary Key is “id” and the Foreign Key is “object_id”.

Attribute	Data Type	Value
“id”	serial	12527
“object_id”	int	208707
“summarisation_text”	text	“District Budgets Probit regression is used to model TA selection for aid on the full sample and linear regression is used to model the quantity of aid received once selected. Budget Allocation (2005-2012) 0.264* 0.191 0.283* 0.0507 0.240 ..... Regime Interactions Probit regression is used to model TA selection for aid on the”
“algorithm_used”	varchar (255)	“textrank”
“created_at”	timestamp	“22022-11-24T14:03:35.544Z”
“updated_path”	timestamp	“2022-11-24T14:03:35.544Z”

Table 11: Sample Database Record for “summarisation” Table

5.3.7 Topic Model Table

Table 12 shows the Topic model table from the database schema along with a sample record. Here, the Primary Key is "id".

Attribute	Data Type	Value
"id"	serial	10
"algorithm_parameter"	varchar (225)	null
"algorithm_description"	text	null
"name"	varchar (225)	"CTM_25"

Table 12: Sample Database Record for "topic_model" Table

5.3.8 Topic Set Table

Table 13 shows the Topic set table from the database schema along with a sample record. Here, the Primary Key is "id" and there are two Foreign Keys – "object_id" and "topic_model_id".

Attribute	Data Type	Value
"id"	serial	2726
"object_id"	int	198619
"topic_term"	varchar (225)	"galaxy,invariant,problem,ritual,water"
"probability"	decimal	null
"topic_model_id"	int	10
"related_object_ids"	varchar (225)	"667,1246,595,1579,1075"
"created_at"	timestamp	"2022-11-27T03:33:26.915Z"
"updated_path"	timestamp	"2022-11-27T03:33:26.915Z"

Table 13: Sample Database Record for "topic_set" Table

5.3.9 Class Table

Table 14 shows the Class table in the database schema along with a sample record. Here, the Primary Key is "id".

Attribute	Data Type	Value
"id"	serial	24
"class_name"	varchar (255)	"Physics"
"classification_scheme"	varchar (225)	"SciBERT"
"depth"	int	null
"class_code"	varchar (255)	null

Table 14: Sample Database Record for "class" Table

5.4 Running the Models for Teams 3 and 4 in our Pipeline

After a few discussions with Dr. Fox, instead of creating APIs for "saving object determination" results and "saving segmentation" results, we took a decision to embed these two models in our backend server¿. This step, in turn, will avoid many API calls and simplify workflows to save the output in both the file system and the database.

We met with each team and discussed the requirements for running the code, the input, the expected output of the model, and the potential difficulties that we might face in achieving these tasks. For Teams 3 and 4, the input is the PDF document for the respective ETD being processed, and the output for both models provided by both teams

is the files and the folder in the ETD directory in the file system. Put simply, this exercise of running the code in the Team 1 code pipeline enables the results to be saved directly in the file system by avoiding making excessive API calls.

An example of the same can be seen in Figure 9:

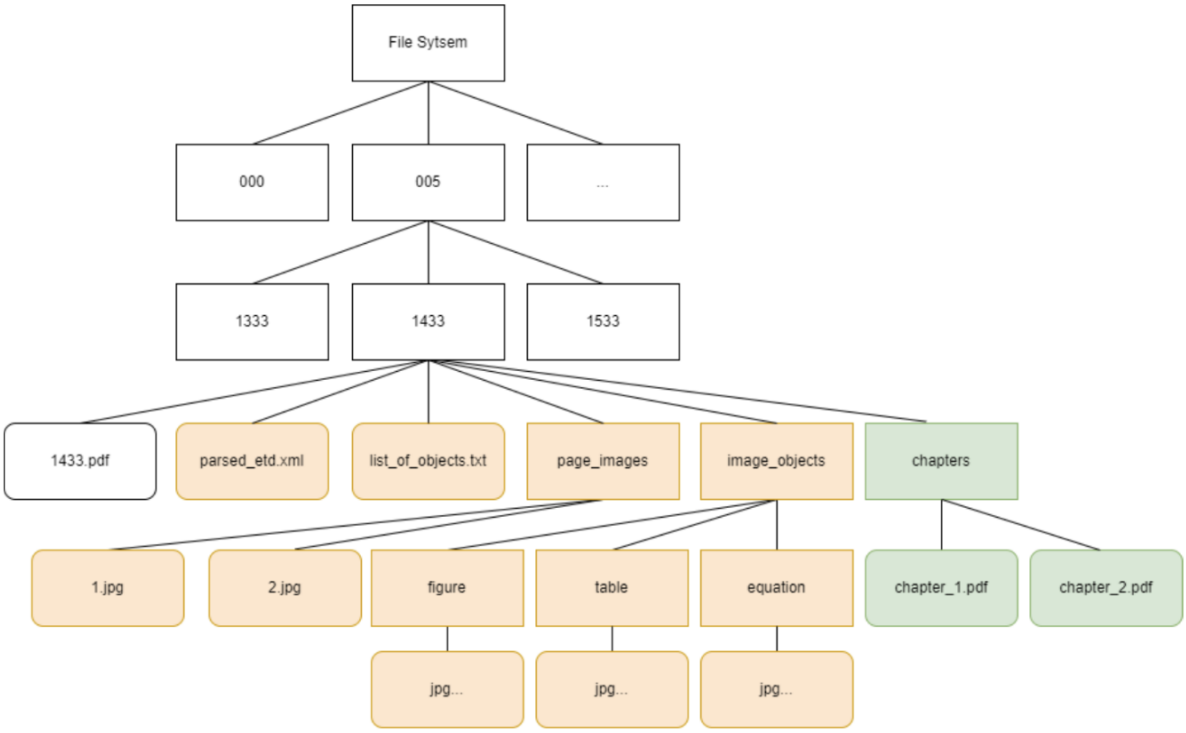

Figure 9: Saved Results in the file system after running the model code in Team 1 container

Except for the file named `1433.pdf`, all folders and files under the ETD with `etd_id` = 0051433 constitute the output from the code for Teams 3 and 4 that we ran on our container. The orange boxes point to the output generated by Team 3's code, and the green boxes point to the output generated by Team 3's code. Our main challenge was the process of storing the files in the file system and recording their relations in the PostgreSQL database at the same time.

For Team 4, we allowed them to run their chapter segmentation code on our file system in the team-1-container-4-GPU container which has a GPU to run the segmentation model and also access to the Camelot server that has the file system. Then we provided them with the get object (`/v1/etds/<int:id>/objects`) and save object (`/v1/objects/<int:etd_id>`) APIs to access and store the results back in the database and file system.

5.5 5K ETD Subset for other Teams

Our database contains over 57,000 ETD records, with 32,000 of them being digital. Due to system limitations and time constraints, the services developed by Teams 3 and 4 were not fast enough to process all of the ETD records. As a result, we prepared a subset of the ETD records that would be supported by the services of all teams.

To create this subset, we first selected the "000" directory in the file system, which contained 8,500 ETDs. We then filtered the ETDs that were digital and published after 2003, resulting in a list of ETDs. From this list, we took the first 5,000 ETDs as the final subset for our analysis. It should be noted that in Table 2, as is explained in Section 5.2.1, the counts for Chapter Records, "Cleaned Text" Records, and Summarization Records represent what

Team 4 produced for those 5,000 ETDs. Likewise, the counts for Object Records, Image Records, Page Records, Text Records, and XML Records are for the subset of the 5,000 that Team 3 was able to process by the time these counts were made.

6 User Manual

Our subsystem has been developed to be accessed completely from the front-end interface. This user manual outlines what information is available to a user and how they can access the repository, and the knowledge needed for working with the tools and frameworks to access the same. Finally, this section will give examples of the data available and what data input and output will look like.

6.1 Front-end Interface Design and Usage

After the submission of Interim Report 1, Dr. Fox suggested that the class form a front-end team, consisting of a representative from each team. During the front-end team meetings, we drew a flow chart to help teams collaborate with each other. In doing so, we decided on the main layout and discussed the overlap of page design. The Team 1 representative, as part of the front-end team, had the responsibility for working on a Curator Front-end UI page leading to the following four pages:

- Statistics Report Page
- Add New ETD Page
- Edit ETD Metadata Page
- Consistency Check Page

Figure 10 shows the wireframe of the Front-end UI for the curator.

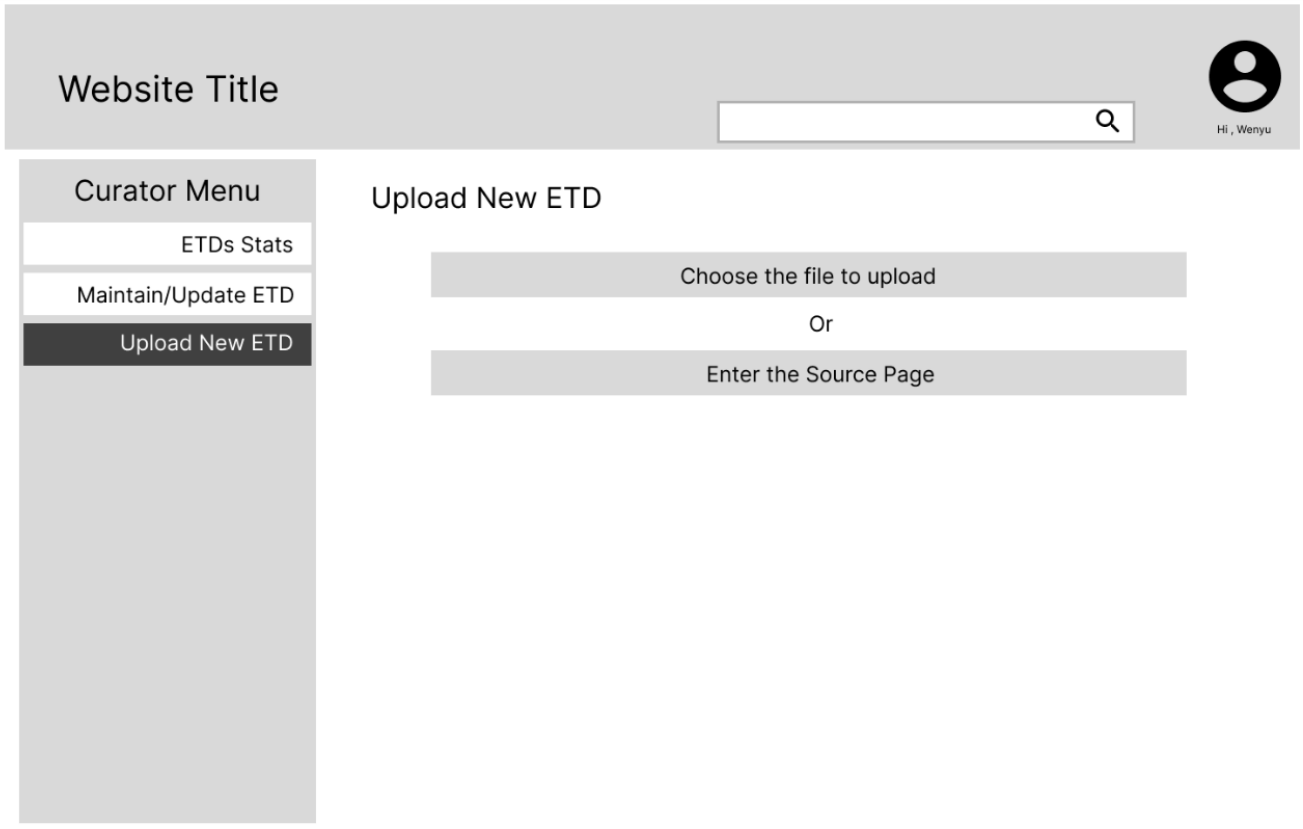

Figure 10: Curator Front-end Page

6.1.1 Current Model

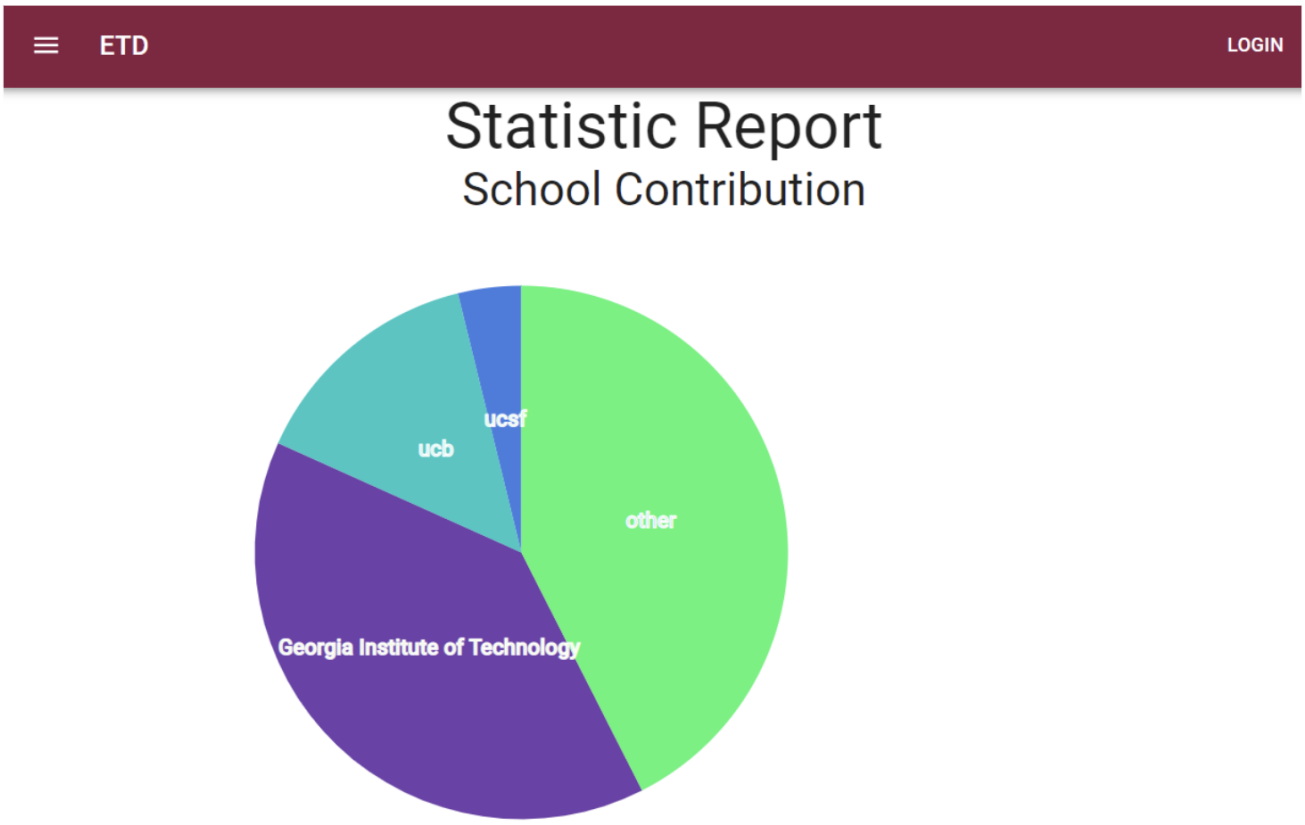

Figure 11: ETD Statistics Page

Figure 11 shows the actual Statistics Report page that has evolved since the creation of the wireframe.

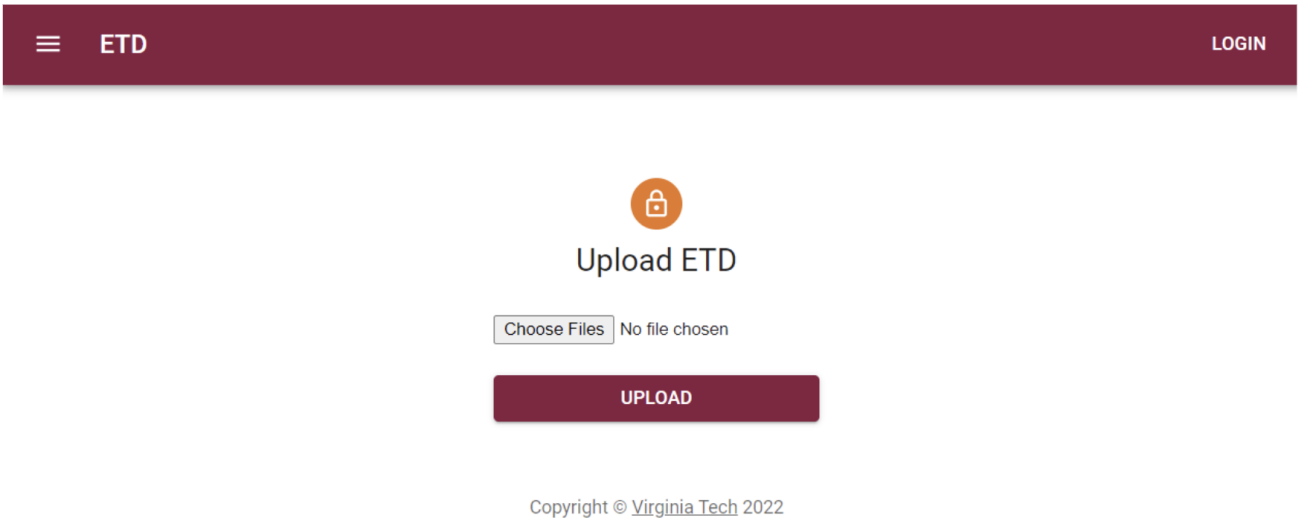

Figure 12: Upload New ETD Page

Another webpage helps the user upload a new ETD, as can be seen in Figure 12. The curator should have the ability to register a new ETD in the database.

6.1.2 Wireframes for Pages Included in Future Work

In the future, additional work should be done on pages for the Curator. For example, future work is needed once it is decided whether we should allow the Curator to add a new ETD by uploading the PDF file, by providing the source page, or both.

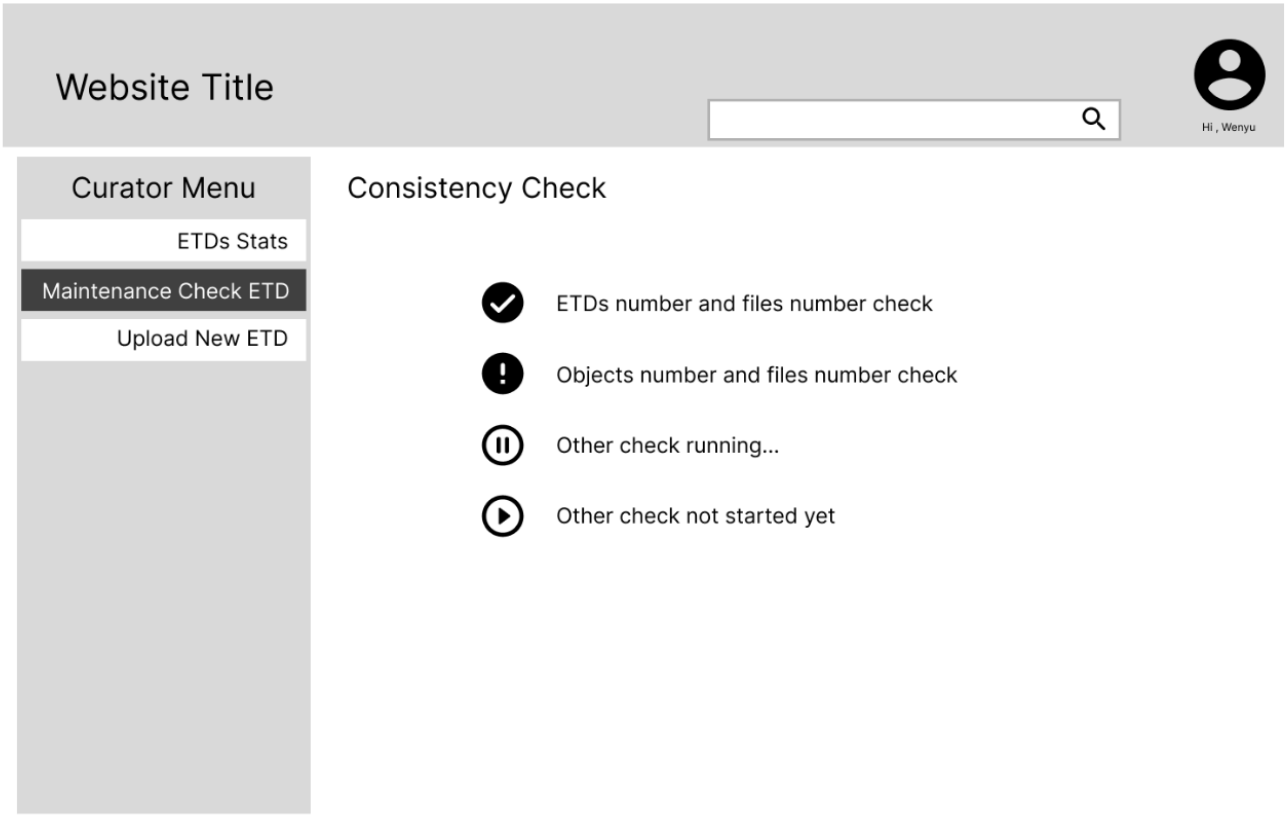

Figure 13: Consistency Check Page

Figure 13 shows how the Curator could run a consistency check for the repository and file system. For example, if the number of ETDs in the file system is different from the number of ETDs in the repository, it would report an error icon. The Curator would then be able to check the error details and fix the problem.

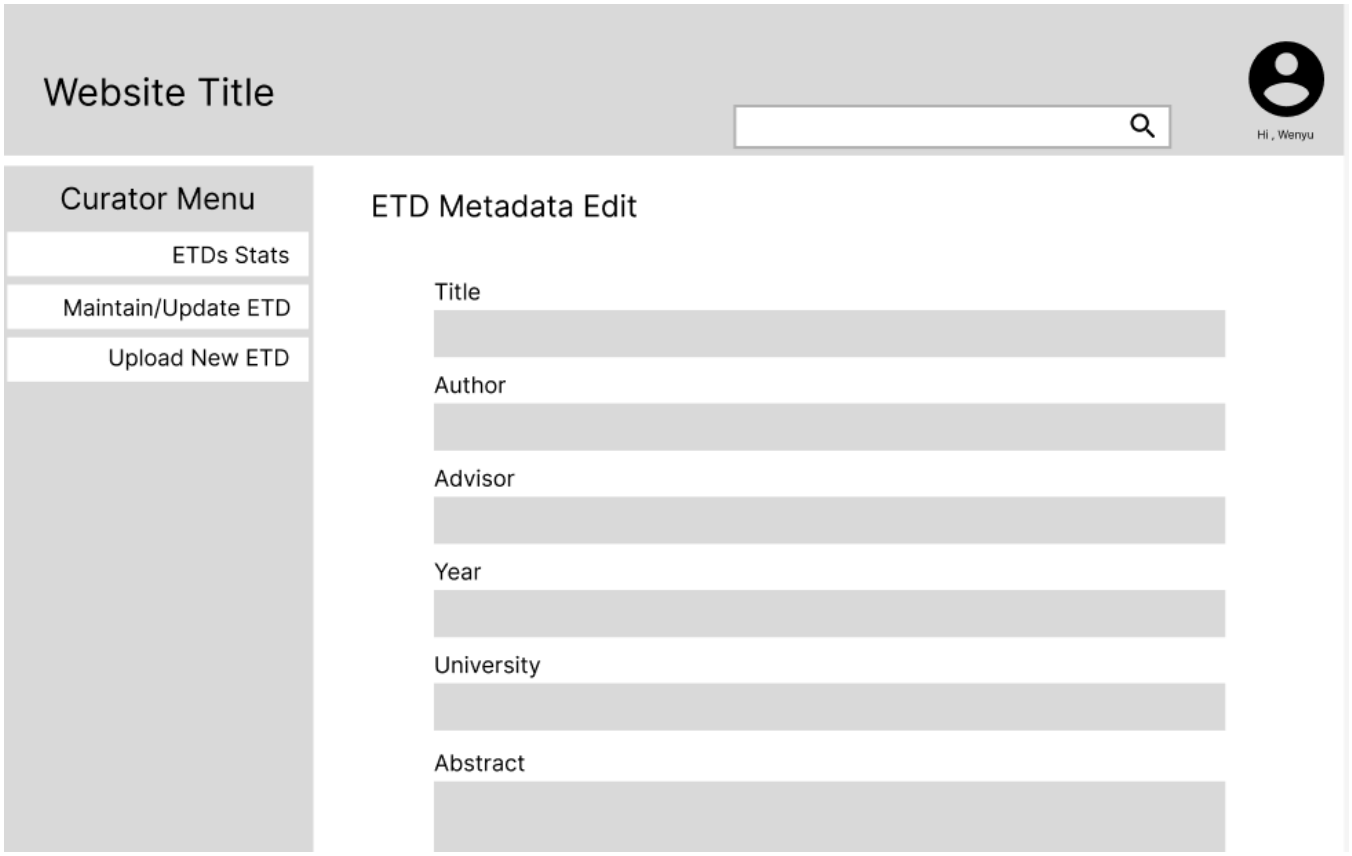

Figure 14: ETD Metadata Edit Page

The Curator could also make simple changes or corrections to the ETD metadata. There will be an "edit" button located on the ETD page which is to be designed by the Team 2 front-end team representative. When the curator searches for a specific ETD or browses the ETDs casually and encounters a problem, he or she could click the button on the ETD display page and it would redirect the Curator to the ETD editing page, as can be seen in Figure 14.

There could also be an additional page that would support batch ETD upload-related functionality or the functionality regarding importing previous data. However, there is need to study the data format and requirements before designing the wireframe for the said web pages.

The front-end website would have a main landing page designed by Team 2, where users can browse and search through various ETDs. The Curator would be able to log into the system to access the advanced usage. After the Curator logs into the system, the first page that would shows would be the Statistical Report page, comprising information such as the percentage of ETDs in different disciplines, the number of ETDs from different institutions, etc.

6.2 Available ETD Metadata

The following is information available for users to view and query against, with a short description of each field.

- ID: The unique identifier for an ETD.
- ETD handle: The persistent link to the ETD.
- Title: The title of the ETD.
- Author: The author of the given ETD.

- Advisor: The advisor of the graduate student who submitted the ETD.
- Year: Year the ETD was submitted/accepted.
- Abstract: The entire abstract for the ETD.
- University: The university to which the ETD was submitted/uploaded.
- Degree: The degree, as a part of which the ETD was written in.
- Department: The department to which the ETD was submitted.
- URI: The web address of the ETD.
- Discipline: The area of study the ETD belongs to.
- Local Path: The path where the ETD is stored in the file system.
- Keywords: List of words or phrases that is useful to get an idea of what the ETD is about.
- Bibliography: The references and citations used in the ETD.

7 Developer's Manual

7.1 Clone GitLab Repository

This section will guide any developer who wants to extend or study the services to set up Team 1's production environment locally.

First, clone the repositories from the following two addresses:

- Back-end server and database: `https://code.vt.edu/lyuze/cs5604-team1-repo`
- Front-end: `https://code.vt.edu/aaron2000/cs5604-front-end`

7.2 Database

7.2.1 Importing the CSV File

In the back-end server repository, there is a file called "csv_parsing.ipynb" that contains the script to transfer the data in the CSV files to the PostgreSQL database. There are two CSV files in the repository and under the CSV folder namely, "df_with_labels.csv" and "merged_withPaths_50k.csv". "merged_withPaths_50k.csv" contains the subset of 57,129 ETDs and ETD metadata from the previous work in this course. It provided us with the local path that indicates the location of each ETD in the file system. "df_with_labels.csv" was provided by Bipasha Banerjee who is the SME for Team 4. It includes a column that indicates which ETD is "born digital" so that we were able to provide the experimenter teams (Teams 3 and 4) to use them. However, the subset that Bipasha gave us is different from the 50k we had, which essentially means that it did not cover all the ETDs that we have worked on. Figure 15 shows the directory structure that includes the CSV files.

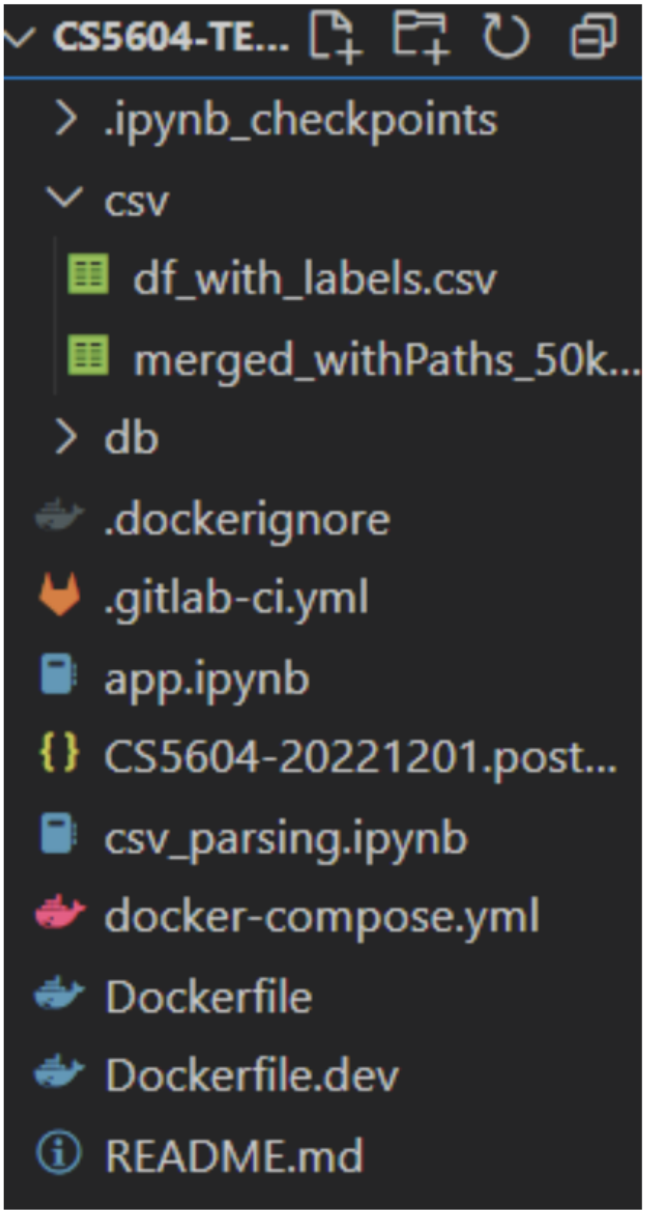

Figure 15: Directory Structure for ETD CSV File

Steps to use the script

- Open the "csv_parsing.ipynb" file.
- Install psycopg2, SQLAlchemy, and Pandas in your environment.
- Check the database connection settings and run all the cells in the notebook. It will load the 57,129 ETD data from the CSV files into the database.

7.2.2 Prerequisites

- To connect to the database remotely, use the database connection settings in the repository. The settings under "# Test DB" are intended to test the database. Use the test database while developing locally and switch back

to the original settings when merging the branch to the main branch.

- To set up a local database (optional, when developers have no remote database access): installation of PostgreSQL is required. The SQL script in the back-end repository, in folder "db", that is named `cs5604.sql`, should be used. Run this script to instantiate a local database. This script generates the initial schema and 57,129 ETD records.

7.3 Back-end Development

The back-end server exposes APIs for other services to access the database, the knowledge graph, and the file system.

7.3.1 Prerequisites

To run the Flask application, the developer needs the following dependencies installed:

- Python 3.9.7
- PostgreSQL database engine with access to create tables in the database
- Jupyter Notebook for developing and running the application
- Flask and Flask SQLAlchemy

7.3.2 Getting Started

Open app.ipynb and run the notebook on the IDE. Once the Flask application runs without errors, the APIs can be accessed by their appropriate routes (e.g., `http://localhost:5000/v2/digitalobjects/all?page=5&per_page=50` will return a list of ETDs and their objects. Find the complete API list in the API section.) It will take a few minutes for the API to generate the response. When you see the output under the last line with the "Running on http://localhost:5000/" message, the server is working as expected.

7.3.3 Code

The file can be divided into four parts:

- Importing: This part imports the necessary libraries and modules like "marshmallow" and "SQLAlchemy".
- Database Connection Configuration: This part contains the database address, name, and password, and then uses these to start the connection with the database.
- Model: This part defines the model class based on the database schema. An example code snippet for a model can be seen in Figure 16.

```python
# Model
class Summarisation(db.Model):
    __tablename__ = "summarisation"
    id = db.Column(db.Integer, primary_key=True)
    object_id=db.Column(db.Integer)
    summarisation_text=db.Column(db.VARCHAR(1024))
    algorithm_used=db.Column(db.VARCHAR(255))
    created_at=db.Column(db.DateTime, default=datetime.utcnow)
    updated_at=db.Column(db.DateTime, default=datetime.utcnow, onupdate=datetime.utcnow)

    def create(self):
        db.session.add(self)
        db.session.commit()
        return self

    def __init__(self, obj_id, summarisation_text, algorithm_used=None):
        self.object_id = obj_id
        self.summarisation_text = summarisation_text
        self.algorithm_used = algorithm_used

    def __repr__(self):
        return f"{self.id}"

#db.create_all()
class SummarisationSchema(ma.SQLAlchemySchema):
    class Meta(ma.SQLAlchemySchema.Meta):
        model = Summarisation
        sqla_session = db.session
    id = fields.Integer(dump_only=True)
    object_id = fields.Integer(required=True)
    summarisation_text = fields.String()
    algorithm_used = fields.String()
    created_at=fields.DateTime()
    updated_at=db.DateTime()
```

Figure 16: Sample Code for the Summarisation Model along with Schema Method Call

- Routing: The logic for all APIs and routing is available here. An example code snippet for routing can be seen in Figure 17.

```python
@app.route('/v1/objects/<int:id>/summarisation', methods=['POST'])
def add_summarisation(id):
    summarisation_schema = SummarisationSchema()

    json_data= request.get_json()
    # Create new Summarisation
    summarisation = Summarisation(
        obj_id= id,
        summarisation_text=json_data["summarisation"],
        algorithm_used=json_data["algorithm_used"]
    )
    db.session.add(summarisation)
    db.session.commit()
    result = summarisation_schema.dump(Summarisation.query.get(summarisation.id))
    return {"message": "New summerisation added.", "summarisation": result}
```

Figure 17: Sample Code for Routing

The most important parts of the code files are the models and the API routes. To interact with the database using SQLAlchemy and marshmallow, we had to design the model according to the database schema. The two aforementioned code snippet screenshots, shown in Figures 16 and 17, illustrate an API interacting with the "summarisation" table in the database.

To create an API, we needed to define the route by using "@app.route('/route')" notation on the POST method, following which we wrote the logic within the same method.

7.3.4 API Testing

We use Postman to help design, organize, test, and share API contracts that are implemented on the Flask server. The API collection that we created can be found in the back-end repository. It is a JSON file called "CS5604-20221201.postman_collection.json". To view the whole collection properly, install Postman locally or create a Postman account and operate it on the webpage. Next to the workspace, click the import button to import the JSON file. See Figure 18.

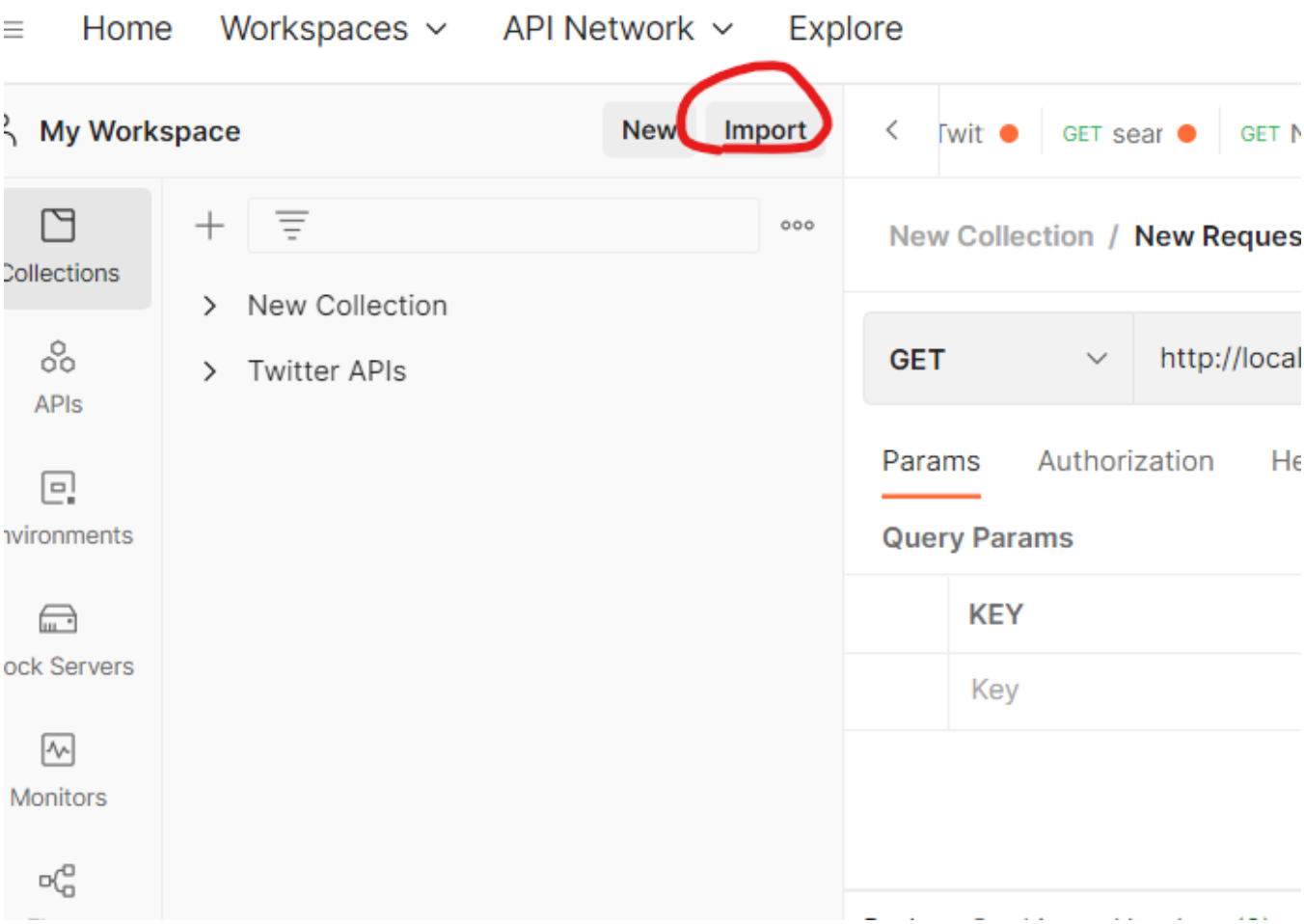

Figure 18: Import Postman Collection

7.4 Front-end Development

The front-end repository is owned by Team 2. It uses React.js as the front-end framework. We contributed two pages and one component, which are relevant to the Curator. Figure 19 shows the location of all these files in the front-end repository.

1. The UploadETD.js page provided upload functionality and sent the request to the "Add new ETD" API that was created at our back-end server.
2. The StatisticReport.js page called the "Get statistics report in school contribution amount" API to get the report number and display it on the page.
3. The PieChart.js component is used by StatisticReport.js, allowing it to show the number in the pie chart. The component can be reusable, which means one page can display several pie charts but only needs one component.

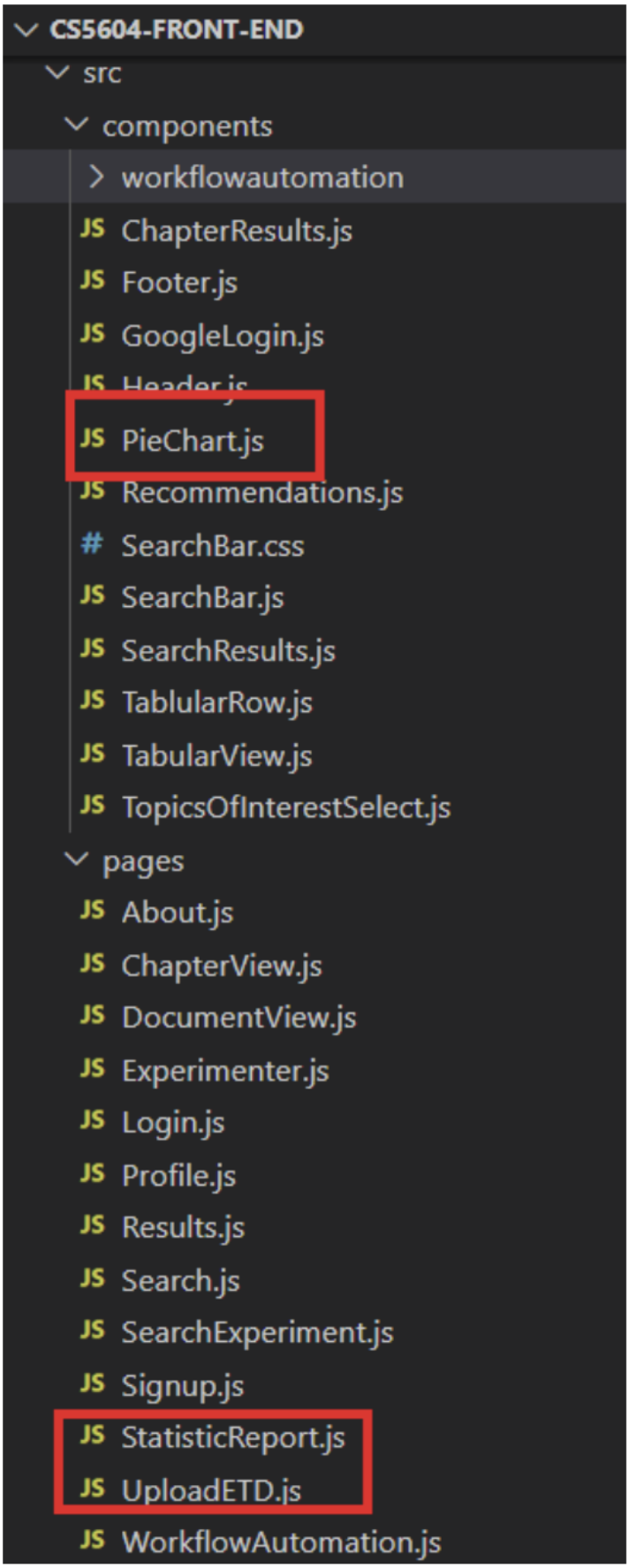

Figure 19: Front-end Modules

7.5 Architecture

Currently, Team 1 enables the connection between the server and the database. The server contains several HTML templates that provide a user interface for end users for simple interactions. With the user interface pages, users can perform CRUD actions on the database safely. Regarding the high-level architecture, Figure 3 gives the reader a fair idea of the various tools and architectures that are being used.

7.6 Hosting the Service on the Cloud Container

We host the Flask application that provides the APIs in a container on the Computer Science Cloud Server (cloud.cs.vt.edu). Team 1 uses 5 containers: "team-1-container-python-rw", "team-1-container-virtuoso", "team-1-container-3-db", "team-1-container-4-gpu", and "team-1-container-jupyter". All five of these containers are present in the DLRL project folder of the discovery cluster.

1. The first container (team-1-container-python-rw) is where the Flask application is running the APIs. This container also has the file system mounted to it.
2. The second container (team-1-container-virtuoso) is where the knowledge graph that Team 2 uses is present. This container has Virtuoso installed, which facilitates storing the RDF triples for the knowledge graph.
3. The third container(team-1-container-3-db) is where the PostgreSQL database is present. When the API is called, it accesses the database present in the third container and returns the results.
4. The fourth container (team-1-container-4-gpu) has the file system mounted, and has been created for Team 4 to run their chapter segmentation model and save the results to the database and file system.
5. The fifth container (team-1-container-jupyter) has Jupyter running, which allows us to create notebooks to make changes to the file system manually if necessary.

The current code is present in the CS5604-Team1-repo created by Team 5 on GitLab (`https://code.vt.edu/lyuze/cs5604-team1-repo0/`). Create a new branch to work on and make changes to the code. The code for the APIs is present in the notebook (app.ipynb). Once the code is ready to be merged to the main branch, send a merge request to the main branch, and assign as a reviewer any one of the maintainers of the project. This information can be found on the project members page on GitLab. The CI/CD pipeline gets triggered once the maintainer reviews the merge request and merges the branch. The pipeline creates the Docker container and builds it. Once the build is complete, the container is deployed onto the cloud. This creates a Flask application and can be accessed at `https://team-1-flask.discovery.cs.vt.edu/<api>`.

7.7 API Contract

7.7.1 Initial APIs

We first created a set of initial APIs to deploy on the cloud.cs.vt.edu server. The purpose of the initial APIs was to test deployability and usability on the side, while the required APIs were under development. We defined APIs that we had identified after consolidating requirements from different teams. The following initial APIs were defined with reference to the requirements previously defined by Team 2. The response is fit to match the data required to index the ETD data for information retrieval. There were hiccups related to the classification and summarization results. The database schema was not compatible to get the results, so we decided to redefine the schema for the classification table, and to leave that part out from the API response.

7.7.2 Background on APIs

The Flask web microframework is utilized to create RESTful APIs in Python. We selected this framework because it is easy to use, efficient, and has the ability to scale up to complex applications. We use a modeling schema for the API. With database schema classes, we were able to ensure that the developed API matches the schema and meets the use cases.

```
app.run(host='0.0.0.0', port=5000)
```

The code mentioned above runs the Flask application. The host here is the server on which we want our Flask application to run. We use 0.0.0.0 which specifies all IPv4 addresses on the local server. This would ensure that the server can be reached from all addresses. The default port number is 5000. The API returns JSON serializable output. We use "jsonify" to make the JSON output serializable. This function wraps "json.dumps()" to turn the JSON output into a response object with "application/json" MIME type.

7.8 Accessing the APIs

The APIs are deployed on the server and can be accessed in the following ways:

We can access it through the browser at `https://team1flask.discovery.cs.vt.edu/<api_path>`.

Here, api_path specifies the path to the API that the user wants to access.

One could also use the following code snippet to access the API using Python.

```
#Code to access the API
 def get_user_data(api, parameters):
      response = requests.get(f"{api}", params=parameters)
      if response.status_code == 200:
          print("sucessfully fetched the data with parameters provided")
          formatted_print(response.json())
      else:
          print(f"Hello person, there's a {response.status_code} error with your request")

 def formatted_print(obj):
      text = json.dumps(obj, sort_keys=True, indent=4)
      print(text)



 #Access user data
 import requests
 import json
 parameters = {
          "username": "kedark"
               }

 get_user_data("https://team1flask.discovery.cs.vt.edu/v1/metadata/all",parameters)
```

7.8.1 API List

Since the other teams would not have direct access to the database and the file system, we supported them by providing APIs depending on their needs. The contract for the APIs for other teams is finalized and discussed below.

- Curator webpage: The APIs allow the Curator to access the ETDs and perform basic operations like create, read, and update.
- Accessing file system: The APIs provide access to digital objects in the file system. The upload API will return the absolute path in the response.
- Accessing SQL database: The APIs would help other teams to save their results in the SQL database and access existing ETD data.
- Accessing RDF triplestore: The APIs would help the end-user team to build a knowledge graph over the RDF triplestore.

Purpose	Method	URL	Description	Team
Get Statistics Report of School Contribution Amount	GET	/v1/report/school-contribution	This API provides statistical reports of the stored ETD collections; intended mainly for front-end page use.	1
Get ETD and its Object (version 1)	GET	/v1/digitalobjects/all?start=1 &limit=20	This API returns all ETDs along with their objects. The "start" and "limit" parameters define the pagination function.	2, 3, 4
Get ETD and its Object (version 2)	GET	/v2/digitalobjects/all?page=5 &`per_page=50`	An improved-performance version of the previous API.	2, 3, 4
Get ETD PDF by etd_id	GET	/v1/etds/<etd-id>/pdf	This API returns the ETD PDF by etd_id.	2, 3, 4
Get ETD by etd_id	GET	/v1/etds/<etd-id>	This API returns an ETD corresponding to an etd_id.	2, 3, 4
Get Objects by etd_id	GET	/v1/etds/<etd-id>/objects?type=chapter	This API returns all object details that belong to the ETD.	2, 4
Save Summarization	POST	/v1/objects/<object-id>/summarization	This API should take a JSON object with two fields and values, and insert into the summarization table with the associated object_id.	4
Save Classification	POST	/v2/objects/<object-id>/classification	This API should take a set of labels and the model that is used. Insert into the classification-related tables, and create a relation between object and class (`object_classification` table). Here, we have label name = `class_name`.	4
Get Object Relations Present in the Knowledge Graph by etd_id	GET	/v1/graph/<etd-id>/	This API returns object relations present in the knowledge graph.	2
Get Object File by object_id	GET	/v1/objects/<object-id>/file	If the type of the object is an image, this API will return the image file.	2
Save Object Topic	POST	/v1/objects/<object-id>/topics	This API should take a JSON body with a set of topic terms and the probability of each term. This saves it in the relevant topic-related table.	3
Get ETD Page Image	GET	/v1/etds/<etd-id>/<page-number>	This API should return the specific page of the ETD.	3

Add New ETD	POST	/v1/etds	Add a new ETD to the database and upload the file to the file system.	1
Save Detected Object	POST	/v1/objects/<etd-id>/<type_name>	This API should take a PDF or image file, save it into the file system under the associate ETD directory, and insert a new row into the associated "object" metadata table with the path in the file system and object type. If successful, the output will be a response consisting of the object_id.	3, 4
Save Detected Objects	POST	/v1/objects/<etd-id>	This is the batch-upload version of the "Save Detected Object" API and takes a zip file and a list of objects in JSON format.	3, 4
Save ETD Page Images	POST	/v1/etds/<etd-id>/page-images	This API saves the generated page images for a particular ETD.	4
Save Topic Modelling Results	POST	/v1/topic_model/object/<object-id>	This API saves the generated topic modeling results.	3

7.9 Developing with Virtuoso

In this section, we will show how to set up a knowledge graph using Virtuoso.

7.9.1 Setting Up

We are using Virtuoso to create the RDF Triple Store. Virtuoso can be downloaded from their official website, which has links for their Open Source Edition for Linux, Windows, MacOS, and Docker images. We have set up the RDF on our local machine and we are waiting for Team 5 to set up a container that has the Virtuoso Docker image.

7.9.2 Loading Knowledge Graph into Virtuoso

We need to save the Turtle document described in Section 4.6 as a `.ttl` file. After starting the Virtuoso server on our local machine, we open the web interface on `http://localhost:8890` and upload the graph using the Quad Store Upload section within the Linked Data tab. Figure 20 shows the process of loading the KG into Virtuoso.

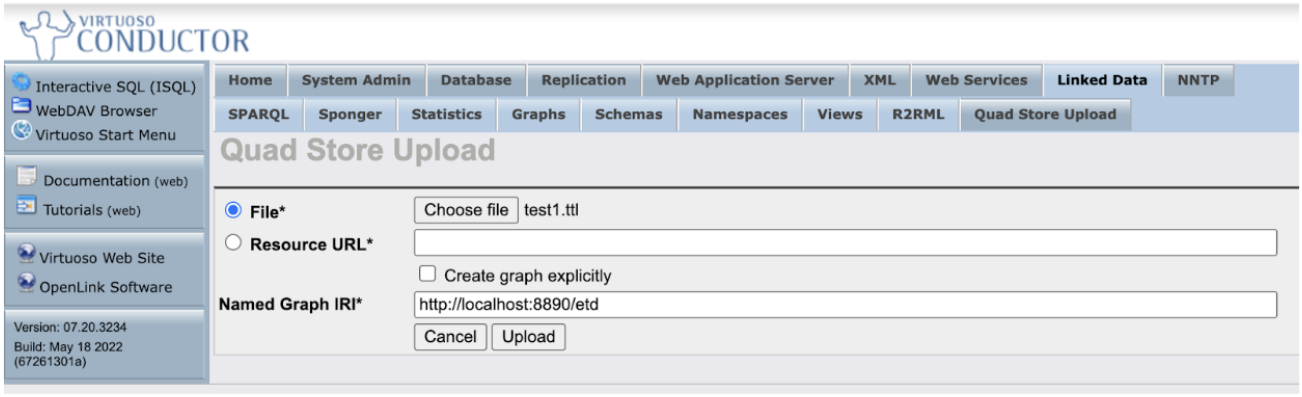

Figure 20: Loading Knowledge Graph into Virtuoso

7.9.3 Querying Knowledge Graph in Virtuoso

We can query the knowledge graph using Linked Data and then navigate to the SPARQL section by providing our Graph Identifier (IRI). SPARQL (pronounced 'sparkle') is the standard query language for accessing RDF triplestores. We have created an API that returns all the relations with the digital objects related to an ETD.

As of now, we have published the API with some sample data present in the knowledge graph. The actual data would be inserted into the knowledge graph when we run the object detection models. Figure 21 contains a snippet of the response.

```
// GET /v1/graph/{etd_id}
{
  "relations": [
    {
      "subject": {
        "id": 2,
        "path": "00/001/chapters/chapter1.pdf"
      },
      "relationship": "has_a_figure",
      "object": {
        "id": 3,
        "path": "00/001/figures/figure1.png"
      }
    },
```

Figure 21: Virtuoso Response

7.10 Future Work

With the current system design, all services have been integrated with our back-end service through APIs to access the repository. During the course, all the teams finalized a subset of ETDs that everyone would work on and perform the operations of topic modeling, object detection, chapter segmentation, and summarization, and finally index the results. If a new ETD is added to our system, we would have to perform all these operations on it, which indicates we need a workflow for it. To make our system production ready, it should support a high volume of incoming data and edits on existing data. Supporting these requirements is not feasible by simply using single operation APIs. We recommend exploring data transfer using messaging queues, which would provide the following advantages.

- Efficiency: Messages are typically added to a queue and then processed asynchronously; they can allow for more parallel processing of data, which can make for more efficient use of resources.
- Reliability: When using an API, if a system component fails or becomes unavailable, the API call will fail, which can cause problems for the rest of the system. With a messaging queue, messages can be stored in the queue and delivered to the recipient at a later time, which can make the overall system more resilient to failures.

One potential direction for future work is to integrate the Knowledge Graph with the Search service's question-answering module. Currently, we have encountered permission errors in Virtuoso that prevent us from performing write operations. If these issues can be resolved, we can implement logic to save information to the Knowledge Graph at the same time that data is inserted into the relational database. This would allow us to create SPARQL queries that can answer common types of questions.

We built a user interface module that facilitates uploading a new ETD and a web page where we display the distribution of ETDs produced at the university level. In the future, the following features should be added to support the Curator.

- Statistics Page: The Curator should be able to see detailed statistics on the information present in our database.
- ETD Metadata Edit Page: The Curator should be able to modify the current ETD Metadata.
- Consistency Check Page: The Curator should be able to compare and validate digital objects extracted from different algorithms.

Another direction for future work is to enhance our Curator service by adding the following components:

- Authenticate Clients: The Curator Service should perform API authentication to verify that a request is coming from a trusted client. This is important because it helps prevent unauthorized access to the API and the data it manages.
- Metrics Dashboard: Developers should be able to track different metrics related to API error growth, CPU and disk usage, etc., and set alerts for them.

In addition to enhancing our Curator service, future work could also involve further expanding the types of information we store about an ETD. Currently, we only store the document link for an ETD in the `etd_metadata` table. However, to provide a more comprehensive experience for users, we could consider storing additional information such as the URL to the landing page of the ETD, as well as links to any supporting documents that may be associated with it. This would allow users to easily explore and access all relevant information related to an ETD, directly through our API.

Additional other data could be stored if provided by other teams. For example, associated with a list of topics assigned to a document or document part by Team 3, there could be a probability value for each of the topics in the list, since that can be determined when inferring what topics should be in the list. Other probability values could be stored with other data, when there is some uncertainty resulting from predictions.

References

[1] Michael Bayer. "SQLAlchemy". In: *The Architecture of Open Source Applications Volume II: Structure, Scale, and a Few More Fearless Hacks*. Ed. by Amy Brown and Greg Wilson. aosabook.org, 2012. URL: http://aosabook.org/en/sqlalchemy.html.

[2] Tim Conrad. *Postgresql vs. mysql vs. commercial databases: It's all about what you need*. 2006.

[3] *Docker*. [Online; accessed 16-September-2022]. 2013. URL: https://www.docker.com/products/docker-hub/.

[4] *Elasticsearch*. [Online; accessed 16-September-2022]. 2010. URL: https://www.elastic.co/elasticsearch.

[5] Orri Erling and Ivan Mikhailov. "Virtuoso: RDF support in a native RDBMS". In: *Semantic web information management*. Springer, 2010, pp. 501–519.

[6] *Jira*. [Online; accessed 16-September-2022]. 2002. URL: https://jira.atlassian.com/.

[7] Ora Lassila, Ralph R Swick, et al. *Resource description framework (RDF) model and syntax specification*. 1999. URL: https://www.w3.org/TR/1999/REC-rdf-syntax-19990222/.

[8] Ian Miell and Aidan Sayers. *Docker in practice*. Simon and Schuster, 2019.

[9] *PostgreSQL*. [Online; accessed 16-September-2022]. 1996. URL: https://www.postgresql.org/.

[10] Kunal Relan. *Building REST APIs with Flask: Create Python Web Services with MySQL, 1st ed.* Apress, 2019. ISBN: 978-1484250211.

[11] *Turtle*. [Online; accessed 04-December-2022]. 2014. URL: https://www.w3.org/TR/turtle/%5C#language-features.

[12] *Vertabelo*. [Online; accessed 04-December-2022]. URL: https://vertabelo.com/documentation/.

[13] *Virtuoso*. [Online; accessed 16-September-2022]. 1999. URL: https://virtuoso.openlinksw.com/.

[14] VTechWorks. *ETDs: Virginia Tech Electronic Theses and Dissertations*. [Online; accessed 16-September-2022]. 2022. URL: https://vtechworks.lib.vt.edu/handle/10919/5534.