# VOiC: Virtual Office in the Cloud

Hayden Carroll*
carr5440@vandals.uidaho.edu
University of Idaho
Moscow, Idaho, USA

Austin Kugler*
kugl5443@vandals.uidaho.edu
University of Idaho
Moscow, Idaho, USA

## ABSTRACT

Digital documents form the backbone of modern office work. This makes a robust document management system an essential part of employee productivity. However, no document management system with intuitive data discovery tools exists. Furthermore, users need access to many formatting tools to write documents for changing needs. An access control architecture must exist to secure these documents and allow their sharing with organizational divisions. This paper presents such a system as a "virtual office in the cloud." We propose a novel search method, rich text editor, and access control scheme.

## CCS CONCEPTS

• **Applied computing** → **Document management**; **Document searching**; • **Computer systems organization** → *Architectures.*

## KEYWORDS

document management, virtual office, access control, graph isomorphism, search algorithms

**ACM Reference Format:**
Hayden Carroll and Austin Kugler. xxxx. VOiC: Virtual Office in the Cloud. In *Proceedings of University of Idaho (Spring Semester).* ACM, New York, NY, USA, 8 pages. https://doi.org/xx.xxxx/xxxxxxx.xxxxxxx

## 1 INTRODUCTION

Events of recent history, namely the COVID-19 Pandemic, accelerated the growth of remote work as an alternative to traditional office settings. Approximately half of workers employed before the Pandemic were working from home in May 2020 [3]. Effective remote work inherently demands a robust web-based environment that provides functionalities previously available in physical offices. Such an environment is especially vital for legacy or small organizations that are reliant on physical storage mediums. This also applies to companies in developing countries. These organizations can support remote work by transferring their offices into the web; that is, becoming a Virtual Office in the Cloud (VOiC).

*Both authors contributed equally to this research.


*Spring Semester, 2022, Moscow, ID, USA*

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/xx.xxxx/xxxxxxx.xxxxxxx

### 1.1 Contributions

In this paper, we propose three core features that define a virtual office:

(1) VOiC provides users with a powerful search tool for data discovery. Each document's relevance in the system is clear due to its assignment of a search graph. This graph provides a logical path that associates the document with any categories it belongs to. Once added, the graph is searchable using principles of graph isomorphism or other methods. Additionally, VOiC provides a more standard text-based searching option.
(2) VOiC permits create, read, update, and delete (CRUD) actions for simple to complex documents through its use of HyperText Markup Language (HTML) for document representation. This allows a wide range of stakeholders to benefit from the system, from those writing simple note documents to formatting-heavy technical papers. The use of HTML also allows a search graph to be directly embedded into a document. As a result, a single entity represents both the document's content and metadata necessary for a graph-based search.
(3) VOiC secures documents with role-based access control while still allowing document sharing explicitly through the username of each user. VOiC's presence in the cloud makes access control an indispensable part of the architecture to prevent damage malicious parties.

## 2 RELATED WORK

Improving the office experience through a feature-rich virtual system is not a new idea. Designs for electronic office systems existed as early as 1985 [8]. By 1996, researchers were already considering the need for a document management system that allows collaborative work [2]. Recent advancements in cloud computing and new technologies allow significant improvements on early architectures for virtual offices. Later systems propose the metadata tagging of data and focus on document version control [6], design blockchain-based architectures to prevent data silos and protect stakeholder access [4], and present role-based sharing in a modern web application [9].

VOiC builds the ideas of metadata tags, protection of stakeholder access, role-based sharing, and use of a modern web application. However, VOiC expands on such systems through its focus on data discovery through graph-based searching and its use of HTML to represent documents. Additionally, VOiC's design is platform-agnostic due to its status as a cloud application. We also use a modern front-end library to ensure a smooth experience on devices of all kinds.

# 3 BACKGROUND

## 3.1 Flask

Flask is a microframework that provides developers with tools useful for building web applications [5]. The framework is minimalist in nature with the core library remaining small. Its two core components include Werkzeug for web server functions and Jinja for HTML templating [7]. Flask is extendable due to its support of many extension libraries. It also works with the majority of third-party Python libraries. Some useful additions for a virtual office include:

- `flask_bcrypt`[1]: BCrypt is a hashing utility useful for securing sensitive information. We use it for password encryption.
- `flask_ckeditor`[2]: CKEditor is an embeddable rich text editor with full support for HTML editing. This extension allows a core feature of VOiC—HTML editing and embedding of graphs directly into documents.
- `flask_login`[3]: Login provides user session management tools. It stores the current user's information, restricts operations to logged-in users, and secures user session data. We use Login for access control and general user management.
- `flask_mail`[4]: Mail is useful in setting up the simple mail transfer protocol (SMTP). We use this extension to email users with a link to reset their password.
- `flask_mobility`[5]: Mobility enhances web app optimization for mobile devices. We use Mobility to tweak front-end rendering for mobile devices.
- `flask_sqlalchemy`[6]: SQLAlchemy is an object-relational mapper (ORM), allowing the transfer of data from relational databases to Python objects. This extension is vital in our architecture as a document management and storage system necessarily requies frequent database access.
- `flask_wtf`[7]: WTForms renders input forms and then validates them upon submission. This extension also provides protection against Cross-Site Request Forgery (CSRF) attacks. We use WTForms for all of these purposes.
- `itsdangerous`[8]: Allows retrieval of data after transmission to an untrusted environment. In VOiC, we use this module to allow users to reset their password through a temporary web address.
- `markupsafe`[9]: MarkupSafe aims to mitigate injection attacks while rendering user inputted HTML. We use it for this feature.
- `bs4`[10]: Beautiful Soup is a convenient library for parsing HTML tags. In VOiC, we use this library to automatically generate missing tags in user input.

[1] https://flask-bcrypt.readthedocs.io
[2] https://flask-ckeditor.readthedocs.io
[3] https://flask-login.readthedocs.io
[4] https://pythonhosted.org/Flask-Mail
[5] https://flask-mobility.readthedocs.io
[6] https://flask-sqlalchemy.palletsprojects.com
[7] https://flask-wtf.readthedocs.io
[8] https://itsdangerous.palletsprojects.com
[9] https://palletsprojects.com/p/markupsafe
[10] https://beautiful-soup-4.readthedocs.io

## 3.2 Bootstrap

Bootstrap[11] is an open source front-end framework for creating platform-agnostic and responsive websites [1]. It provides a wide range of styling options using Cascading Style Sheets (CSS). We use Bootstrap 4 to ensure VOiC is responsive, has an attractive user interface (UI), and works on many web browsers on different devices.

One of Bootstrap's major contributions is containers. Developers define containers with the `<div>` HTML tag. Each container represents a distinct part of a webpage. They allow for centering, padding, and application of CSS styles to an entire section of content.

Bootstrap also contributes a 12 column layout to create a grid system. Individual components of a website are re-sizeable by changing the number of columns they occupy, from 1 to 12. In VOiC, we use the grid system to ensure the document viewing experience remains intuitive on all platforms.

We also use many of the custom CSS styles provided by Bootstrap: `form-group` for forms; `btn` and `btn-outline-info` for buttons; `navbar-nav`, `nav-item`, and `nav-link` for the navigation bar; `media content-section`, `media-body`, and `article-metadata` for document viewing; and `text-muted` for footnotes, to name a few.

## 3.3 SQLAlchemy

SQLAlchemy is a Python-specific object-relational mapper (ORM). It seamlessly converts data from a relational database into Python objects. Therefore, there is no requirement for a specific relational database model. This service is useful in VOiC as we can use an SQLite database for development and easily transition to an SQL database for deployment.

We use SQLAlchemy to define our models and tables for the database, as well as perform all queries for the project. Models created using SQLAlchemy include *User*, *Role*, and *Document*. By defining each model and table, we were able to create a basic, functional database with relationships between different tables. Querying the table also proved to be rather simplistic, since SQLAlchemy creates an interface between Python and the relational database.

# 4 ENVIRONMENT SETUP

Setting up a development environment for VOiC has several prerequisites and requires a few steps. First, developers should use a Unix-based operating system or subsystem. We chose to use the Windows Subsystem for Linux (WSL)[12].

Developers should then install the Python[13] programming language. We use version 3.8.10 but VOiC is compatible with most recent Python versions.

The next step is cloning the VOiC repository from code-hosting website GitHub. The git[14] command-line tool is a prerequisite for this step.

```
$ git clone https://github.com/austinpkugler/voic.git
```

This will copy the project's source code to your local machine.

[11] https://getbootstrap.com
[12] https://ubuntu.com/wsl
[13] https://www.python.org
[14] https://git-scm.com

You will need to create an environment file named `.env` in the `voic/` directory. It should follow the structure:

```
FLASK_SECRET_KEY="anysecretkey"
EMAIL_USERNAME="youremail@email.com"
EMAIL_PASSWORD="youremailpassword"
DATABASE_URL='sqlite:///voic.db'
```

At this point, all dependencies are installable with Python's pip package installer. Simply use the below command while in the `voic/` directory.

```
$ pip install -r requirements.txt
```

VOiC's database is not available on GitHub due to storage limitations. Developers must create it locally using the following command while in the `voic/` directory:

```
$ python create_db.py
```

Installation of VOiC and its dependencies is now complete. The website can be hosted locally using the provided script.

```
$ python run.py
```

It is then usable in any modern web browser at the localhost address, commonly http://127.0.0.1:5000/.

# 5 COMPONENTS

## 5.1 Overview

VOiC provides a comprehensive document manage system—a Virtual Office in the Cloud. It consists of four main processes: storing, sharing, searching, and rendering. Storage includes the holding of users, roles, and documents in a relation database. Sharing deals with access control, allowing users to access pertinent documents through their username and role. Searching serves users with a tool for data discovery with documents searchable by their search graph, title, and content. Rendering forms the front-end portion of VOiC. Together, these elements form a robust solution for document management in a virtual setting.

The full source code is available here[15].

## 5.2 Storage

VOiC uses a relational database to store information essential to its operation. It does so in *User*, *Role*, and *Document* tables. Each contains many attributes.

The *User* table stores system users and includes the following:

- `id`: The unique identification number of the user.
- `username`: The user's preferred unique username.
- `email`: A unique user-provided email used for password reset requests.
- `password`: The user's hashed password.
- `picture`: A profile picture optionally uploaded by the user.
- `created_at`: The UTC time of user creation.
- `roles`: All roles shared to the user.
- `documents`: All documents shared to the user.

The *Role* table defines work roles that users may have. Roles may exist for engineering, human resources, management, and other organizational divisions. It includes:

- `id`: The unique identification number of the role.

[15] https://github.com/austinpkugler/voic

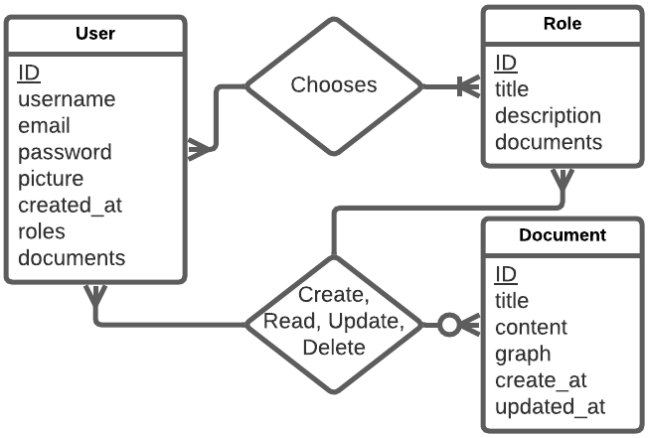


**Figure 1: Entity relationship diagram of VOiC's storage scheme.**

- `title`: The job title of role (e.g. engineer).
- `description`: A brief description of the role.
- `documents`: All documents shared with the role.

https://www.overleaf.com/project/62684ce1b481ca49019427c8 The *Document* table stores all documents in the system and includes the attributes:

- `id`: The unique identification number of the document.
- `title`: The document's title, displayed as a heading.
- `content`: The rich text body of the document including all text, images, tables, and HTML. The document's search graph hidden as part of the content, see figure 2.
- `graph`: The document's search graph, represented as a string. VOiC queries this value during graph searches. We also embed it in the document as part of hidden HTML tags.
- `created_at`: The UTC time of document creation.
- `updated_at`: The UTC time of the last edit made to the document.

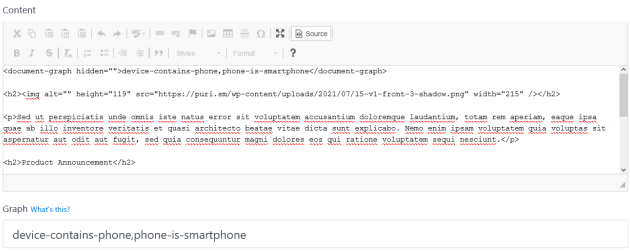


**Figure 2: A search graph embedded in the document's rich text content.**

There exists a many-to-many relationship for *User—Document*, *Role—Document*, and *User—Role*. This follows the logical ideas that one user can access multiple documents, one role can access multiple documents, and a user can have many roles.

Our use of SQLAlchemy allows for the creation of these tables and relationships in relational database of many types. We use

SQLite for local development. We convert the database to use PostgreSQL for deployment.

## 5.3 Sharing

We provide users with document sharing capabilities in two ways. First, users must sign in to authenticate their account before accessing or creating any documents. During document creation, the creator chooses which roles and individual users will have access. This selection is done through two multi-select input fields. By default, all users have the "Employee" role. Therefore, sharing a document with this role shares it to all system users.

After creation, the added users and roles can then edit these permissions as desired. Each user then sees documents they have access to on their dashboard. We generate this list of documents by filtering out each document that does not have the active user in its list of users or any intersection between the document and user roles.

Table 1 shows in which cases a user can access a document. Users must sign in and their username or role added to a document before they can access it.

**Table 1: Conditions for document access by a user.**

| Can Access | Signed In | Shared To |
|---|---|---|
| Y | Y | Y |
| N | N | Y |
| N | Y | N |

## 5.4 Searching

VOiC has two methods of searching for a particular document in the list of documents the current user has access to. The first is the substring search. To perform a substring search in the application, a user types the search query in the search bar and presses submit. A SQLAlchemy query then executes and retrieves all documents in which the search query string is a substring of the title or content. Substring searching is case insensitive and also searches the raw HTML content of the document. Therefore, all HTML tags for a particular document are also searched.

The second method of searching for a document is the graph search. To perform a graph search in the application, a user types "graph:a-b-c,c-b-d." The text after "graph:" is the graph the user wishes to search for. Pressing the submit button executes this graph as a query. Graph's input in the search bar has a specific syntax in the form of vertex-edge-vertex, with commas separating each vertex-edge-vertex group. In the back-end, a query is performed with SQLAlchemy to list all documents where the graph input is a subgraph of the graph in the document.

The general algorithm for this query is as follows:

(1) Alphabetize each vertex-edge-vertex group in both the document and the search graph, so the first vertex in the group is also the first vertex alphabetically. For example, b-c-a would be converted to a-c-b.
(2) Next, iterate through each document and determine if the search graph query is a subset of the document's graph. For a match to occur, the document graph must contain every vertex-edge-vertex group in the search.

## 5.5 Rendering

Rendering is how VOiC displays all front-end content. For styling. we primarily use Bootstrap for this purpose with some custom CSS styles as well. We contain all front-end content in Flask HTML templates and use the Flask renderer to display it.

Each webpage has its own template, with all inheriting from the `base.html` template. This base template contains all universal parts of the web application. For example, library imports, the navigation bar, logo, and footer. The base template also contains a special

```
{% block content %}{% endblock content %}
```

declaration that child templates provide content for.

Templates exist in the `templates/` directory. In this location, we also organize form-containing templates into a `forms/` subdirectory.

Formless templates include `document.html` for document viewing, `help.html` for helpful tips, and `home.html` that contains a dashboard of document previews for the current user.

Pages with forms include `account.html` for update of the user's personal information, `danger-zone.html` for account deletion, and `edit-document.html` for the editing of a document in CKEditor.

For account management, `request-password-reset.html` allows initialization of a password reset request. An additional template, `reset-password.html`, contains an input box for the new password. The `sign-in.html` page lets existing users login, and `sign-up.html` is where account creation occurs.

# 6 STRUCTURE

We follow standard Flask practices for organizing our code. The `__init__.py` file sets environment variables and initializes the application. Database functionality exists in `models.py`, including definitions for the *User*, *Role*, and *Document* tables.

The `routes.py` module provides functions for each page of the website. In these functions, calls to the Flask renderer display pages using HTML templates. The `templates/` directory contains all Flask HTML templates, styled with Bootstrap 4.

All input forms exist in the `forms.py` module. We store data in `voic.db` and provide the `create_db.py` script to create it. Environment variables exist in the `.env` file. A list of VOiC dependencies is available in the `requirements.txt` text file.

## 6.1 __init__.py

In Python, `__init__.py` is a special file in packages. Implicit execution of this file occurs at the time of its containing package's import. In Flask projects the file is useful for performing initialization processes. Its invocation occurs when Flask app is run.

In VOiC the file loads environment variables from an `.env` file, defines configuration values, and creates instances of Flask extension libraries. Other files can then import these settings and instances using the `from voic import *` syntax.

The init file is also where initialization of the Flask application occurs. In VOiC, we name the Flask application object `app`. Importing this object in other modules allows access to config values and is mandatory for route creation.

## 6.2 models.py

The `models.py` module defines the attributes database tables. It also provides utility functions necessary for attribute management. In this module, we define each table in the database using a Python class. Definition of table relationships occurs by adding a special class attribute that uses SQLAlchemy. Flask uses these classes to generate the database.

Note that we place reasonable length limitations on user provided attributes in all classes. We use SQLAlchemy's primitive and abstract data types to define the type of each attribute.

We define a `User` class to define the *User* table in the database. It includes methods for the password reset process.

```
class User
    __tablename__ = "user"
    id = primary key Integer
    username = unqiue not nullable String
    email = unqiue not nullable String
    password = not nullable String
    picture = not nullable String
    created_at = not nullable DateTime
    roles = ManyToMany Role
    documents = ManyToMany Document

    procedure __repr__()
        Return object's String representation

    procedure get_reset_token()
        Generate time sensitive token
        Return token
    endprocedure

    procedure verify_reset_token(token, expires_in)
        Verify token is valid and not expired
        Return User object if valid, None otherwise
    endprocedure
endclass
```

We define user roles in the `Role` class.

```
class Role
    __tablename__ = 'role'
    id = primary key Integer
    title = not nullable String
    description = not nullable Text
    documents = ManyToMany Document

    procedure __repr__()
        Return object's String representation
    endprocedure

    procedure __eq__(other)
        Return True if equal to other, False otherwise
    endprocedure

    procedure __hash__()
        Return unique hash value for object
    endprocedure
endclass
```

We define the *Document* table using a `Document` class.

```
class Document
    __tablename__ = 'document'
    id = primary key Integer
    title = not nullable String
    content = not nullable Text
    graph = String
    created_at = not nullable DateTime
    updated_at = not nullable DateTime

    procedure __repr__()
        Return object's String representation
    endprocedure

    procedure __eq__(other)
        Return True if equal to other, False otherwise
    endprocedure

    procedure __hash__()
        Return unique hash value for object
    endprocedure
endclass
```

## 6.3 routes.py

App routing is a fundamental Flask component. In app routing, a mapping exists between each web address route of the application and a Python function that contains logic for that address. Function decorators declare which route the function belongs to.

Route functions commonly include logic for rendering HTML templates and allowing form submissions. Developers can place limitations on their access using the `flask_login` extension. We secure many of VOiC's routes in this manner. Tables 2 and 3 show which routes are accessible to signed in and signed out users, respectively.

**Table 2: Availability of account-related routes based on user authentication.**

| Route | Signed In | Signed Out |
|---|---|---|
| `/sign-up` | N | Y |
| `/sign-in` | N | Y |
| `/sign-out` | Y | N |
| `/account` | Y | N |
| `/request-password-reset` | N | Y |
| `/reset-password/{token}` | N | Y* |
| `/danger-zone` | Y | N |
| `/delete-account` | Y | N |

*Requires user access to the specific document.

## 6.4 forms.py

Flask forms instruct the renderer to render input fields as a single group. They also supply data validation tools to prevent invalid user input. VOiC includes forms for input related to searching, sign up, sign in, updating account information, performing document

**Table 3: Availability of document-related routes based on user authentication.**

| Route | Signed In | Signed Out |
|---|---|---|
| / | Y | Y* |
| /new-document | Y | N |
| /view_document{id} | Y† | N |
| /edit-document/{id} | Y† | N |
| /delete-document/{id} | Y† | N |
| /duplicate-document/{id} | Y† | N |
| /delete-all-documents | Y | N |
| /help | Y | Y |
| *Rendered differently for signed out users. | | |
| †Requires user access to the specific document. | | |

operations, and resetting a password. Forms use Flask's WTForms extension to ensure consistent rending, access data types designed for input forms, and validate input.

We define each of these forms as follows:

```
class SearchForm
    search_bar = StringField
    submit = SubmitField
endclass

class SignUpForm
    username = StringField
    email = StringField
    password = PasswordField
    confirm_password = PasswordField
    submit = SubmitField

    procedure validate_username(username)
        user = user with inputted username
        if user already exists
            raise username is taken error
    endprocedure

    procedure validate_email(email)
        user = user with inputted email
        if user exists
            raise email is taken error
    endprocedure
endclass

class SignInForm
    email = StringField
    password = PasswordField
    remember = BooleanField
    submit = SubmitField
endclass

class UpdateAccountForm
    username = StringField
    email = StringField
    picture = FileField
    roles = SelectMultipleField
    submit = SubmitField

    procedure validate_username(username)
     if inputted username is not current user's username
            user = user with inputted username
            if user exists
                raise username is taken error
    endprocedure

    procedure validate_email(email)
        if inputted email is not current user's email
            user = user with inputted email
            if user
                raise email is taken error
    endprocedure
endclass

class DocumentForm
    title = StringField
    content = CKEditorField
    graph = StringField
    submit = SubmitField
    roles = SelectMultipleField
    users = SelectMultipleField
endclass

class RequestPasswordResetForm
    email = StringField
    submit = SubmitField

    procedure validate_email(email)
        user = user with inputted email
            if user
                raise no account found error
    endprocedure
endclass

class ResetPasswordForm
    password = PasswordField
    confirm_password = PasswordField
    submit = SubmitField
endclass
```

## 7 SELECTED WEBPAGES

### 7.1 Sign Up

On the sign up page, unregistered users can create a new account. This requires a unique username and unique email. Then, users enter their preferred password in the relevant field and confirm it in the next field. A flash message will appear at the top of the page if the user provides invalid input. After sign up, redirection to the sign in page will occur.

### 7.2 Sign In

On the sign in page, users can login to their previously created accounts. Users must provide their email address and password to do so. If the login information is invalid, a flash message will appear. Users can reset their password from this page by clicking the hyperlink labelled "Reset." After successful sign in, the page redirects to the home page.

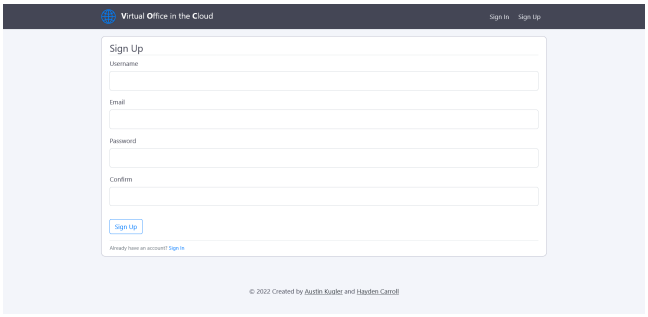


**Figure 3: The page where a new user can create their account.**

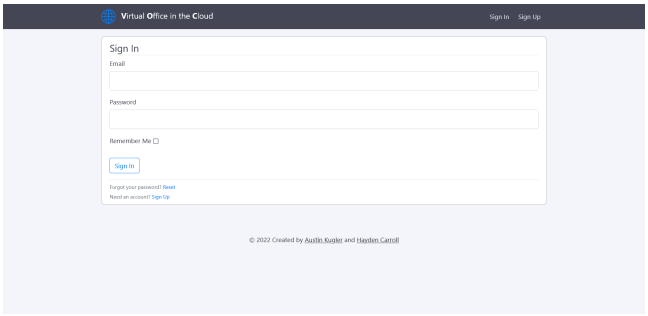


**Figure 4: A page for existing users to login to their account.**

## 7.3 Home

VOiC's home page displays differently depending on the current user's authentication status. If the user is not signed in, the home page is a simple landing page with information about the website.

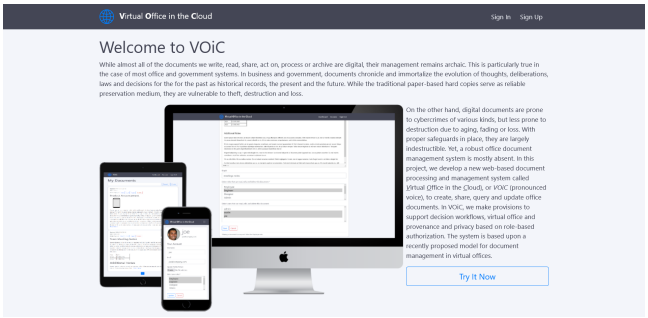


**Figure 5: The home page when the user is not signed in.**

When the user is signed in, the home page displays their dashboard. This dashboard contains a truncated preview for each document they have access to. Each document preview has four buttons: "View," "Edit," "Copy," and "Delete." Pressing the "View" button opens the entirety of the document for reading. The "Edit" button opens an edit document page for the relevant document. "Copy" duplicates the document and appends "- Copy" to the title. "Delete" removes the document from the database completely. At the top of the page, users can utilize the search bar to find documents through a text or graph-based search. There is also a button to create a new document.

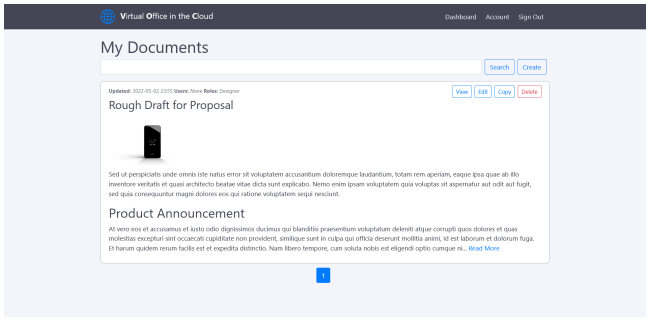


**Figure 6: The home page when the user is signed in.**

## 7.4 Edit Document

The edit document page provides several fields for the user to update a document through. Chaging the input box labeled "Title" updates the document heading, "Content" updates its rich text, "Graph" controls the document's search graph, and the role and user selections control access. After changing these fields, users should press the "Save" button to confirm the changes. This will update the document's attributes in the database and return the user to the home page. A button for deleting the document is also available. We use this same page for document creation.

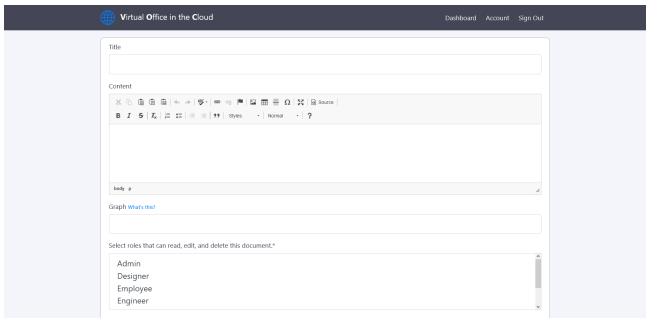


**Figure 7: The page for creating and editing documents.**

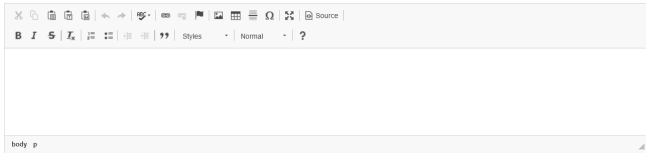


**Figure 8: CKEditor, a rich text editor used by VOiC.**

## 7.5 Account

A user's account page shows their personal information and allows its update. Users can change their profile picture, username, email, and roles through the first form on this page. The second form provides access to the user's "Danger Zone."

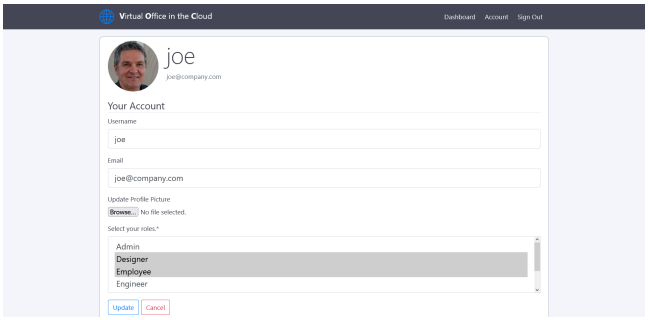


**Figure 9: A user's account page.**

## 7.6 Danger Zone

In the danger zone, users can delete all their documents at once or their entire account. Accessing this page requires navigating through the accounts page. It is purposefully separated into its own page to prevent accidental deletions.

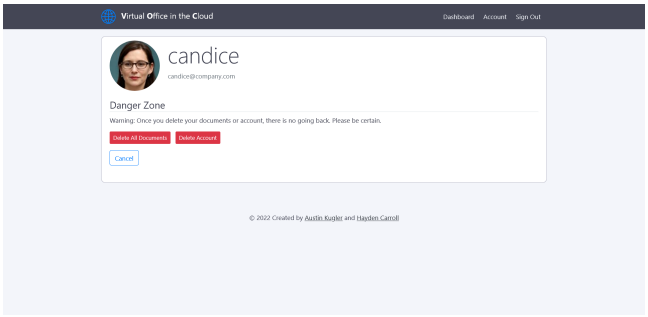


**Figure 10: A user's danger zone page.**

# 8 DISCUSSION

## 8.1 Overhead

Our implementation of graph-based searching has notable overhead over simple substring searching. However, it adds a significant contribution in document organization. Users can define a document's place through a logical progression using a search graph. This provides a more vigorous method for organization and searching, at the cost of increased overhead.

## 8.2 Future Work

In VOiC, search graphs persist in two ways. They exist in the database as an attribute of the *Document* table but also in hidden HTML tags. Saving search graphs in hidden HTML tags allows them to persist in a document even after its removal from the VOiC ecosystem. Currently, we retrieve document search graphs from the database, not from the embedded hidden tags. Future work could eliminate the need for a separate graph attribute and access the graphs from the documents directly. Future research could use the BeautifulSoup library to parse the relevant hidden HTML tags from a given document, acquiring the graph.

Another area for future work is in our graph-based search. Currently, we do not use graph isomorphism as part of our search. Its addition will increase the value of the search feature by providing more accurate results.

The user interface for graph creation and viewing is an area for future work. As of now, graphs are input by users as strings. Implementation of a graphical drag-and-drop interface is more intuitive.

Addition of document read and write permissions will allow for a more secure application. In its current form, VOiC does not provide this feature to users. All users and roles with access to a document can perform CRUD operations on it with no limitations.

Finally, future developers may want to add a special admin role. Presently, role creation occurs during the creation of the database itself. This means there is no straightforward way to add roles after the initial database creation. An admin role will allow specially designated users to create and delete roles as necessary. Additionally, the admin could assign users with their roles. Right now, users are able to change their roles at will.

# 9 CONCLUSION

In this paper, we propose VOiC, a robust system for digital document management. VOiC provides a powerful graph-based search tool for data discovery. It permits CRUD operations on rich text documents. An access control scheme uses both roles and usernames to allow document sharing. We implement VOiC and show its efficacy as a full fledged virtual office. We also describe VOiC's underlying technologies and give instructions on their setup.

# REFERENCES

[1] Suman Aryal. 2019. *Bootstrap: a front-end framework for responsive we design (Bachelor's thesis).* Technical Report. Turku University, Turku, Finland.

[2] A. Backer and U. Busbach. 1996. DocMan: a document management system for cooperation support. In *Proceedings of HICSS-29: 29th Hawaii International Conference on System Sciences*, Vol. 3. University of Hawaii at Manoa, Wailea, Maui, 82–91 vol.3. https://doi.org/10.1109/HICSS.1996.493179

[3] Erik Brynjolfsson, John J Horton, Adam Ozimek, Daniel Rock, Garima Sharma, and Hong-Yi TuYe. 2020. *COVID-19 and Remote Work: An Early Look at US Data.* Working Paper 27344. National Bureau of Economic Research. https://doi.org/10.3386/w27344

[4] Moumita Das, Xingyu Tao, and Jack C. P. Cheng. 2021. A Secure and Distributed Construction Document Management System Using Blockchain. In *Proceedings of the 18th International Conference on Computing in Civil and Building Engineering*, Eduardo Toledo Santos and Sergio Scheer (Eds.). Springer International Publishing, Cham, 850–862.

[5] Miguel Grinberg. 2018. *Flask web development: developing web applications with python.* "O'Reilly Media, Inc.", Sebastopol, CA.

[6] Matthew D Jacobsen, James R Fourman, Kevin M Porter, Wirrig Elizabeth A, Mark D Benedict, Bryon J Foster, and Charles H Ward. 2016. Creating an integrated collaborative environment for materials research. *Integrating Materials and Manufacturing Innovation* 5, 1 (2016), 232–244.

[7] Kunal Relan. 2019. Beginning with flask. In *Building REST APIs with Flask.* Springer, Apress, Berkeley, CA, 1–26.

[8] S. Schindler, U. Flasche, and R. G. Herrtwich. 1985. Electronic Office Systems, International Standards/Recommendations/Specifications for Document Exchange and the Committee Support System. In *Offene Multifunktionale Büroarbeitsplätze und Bildschirmtext*, F. Krückeberg, S. Schindler, and O. Spaniol (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 211–229.

[9] Aryaman Tewari, Ankit Chauhan, and Vishwadeepak Singh Baghela. 2022. *Smart Digital Platform for Text Documents Sharing & Information Retrieval.* Technical Report. EasyChair.