xData-CogA Intern Report

By

Kenneth Gao

Supervisor

Ngoh Wei Jie

Duration

November 2023 – January 2024

Table of Contents

1 Brief Introduction

During my tenure at HTX, I actively participated in two projects: VerText and GenAI. My responsibilities for VerText were centred around refining the frontend codebase, alongside various CI/CD tasks. In the GenAI team, I focused on experimenting with ways to better utilise Knowledge Graphs for Retrieval Augmented Generation (RAG) in conjunction with Large Language Models (LLMs) to advance their capabilities and efficiency. In the subsequent sections, I will provide a comprehensive overview of my contributions to both VerText and GenAI and their outcomes.

2 VerText

VerText serves as a centralized repository housing a comprehensive suite of Natural Language Processing (NLP) tools that encompass a diverse array of functionalities, including question answering, sentiment analysis, summarization, topic modelling, translation, and chat capabilities. The frontend development of VerText leverages the React framework and integrates with the backend through Docker containers.

The development team rigorously adheres to an agile software development framework, with the scrum process serving as a foundational methodology. Integral to this approach are bi-weekly sprint planning sessions and daily stand-up meetings, which form the cornerstone of the agile methodology. This structured and iterative methodology significantly contributes to the efficacy of software project management, providing a stark departure from prior experiences characterized by a lack of defined structure or workflow.

This exposure has profoundly enriched my understanding of software project management, fostering insights into the efficacy of structured methodologies in enhancing project outcomes.

2.1 Description of Work

My journey in VerText began with the completion of the mini assignment, which introduced me to the basics of creating a backend API with Python and Flask, creating a frontend site with React, TS, and antdesign, and tying them all together with Docker containers. I found it particularly enlightening as it provided a practical overview of VerText's architecture.

Following that, I was assigned the responsibility of enhancing the frontend codebase, with tasks ranging from augmenting the CI/CD pipeline to refactoring frontend components for greater reusability.

The improvements and modifications made during this phase were successfully integrated into the main repository of VerText.

2.1.1 Synchronisation of GitLab and Jira – CI/CD

Jira is a project management tool designed to streamline and optimize various aspects of software development. It facilitates agile methodologies, offering features for efficient issue tracking and collaboration. On the other hand, GitLab serves as a Git repository manager, allowing teams to effectively manage source code. Beyond version control, it encompasses features such as CI/CD and code review, providing an all-in-one solution for the development lifecycle.

To propose changes to the VerText codebase, a developer would typically create a new branch, make modifications, and then create a Pull Request (PR) for the changes to be reviewed and merged into the main codebase. Typically, reviewing such changes is time consuming and should be logged on Jira as a task so that stakeholders can gain more insight into the team's progress and developers can block out time to review changes, preventing them from taking on too much work.

However, as many PRs are created in a day, it can be time consuming to create a new task on Jira for each and every PR. Hence, it would be beneficial to automate the process of creating Jira tasks for PRs. Unfortunately, as HTX uses on-prem Jira and GitLab, we are unable to make use of GitLab’s Jira integrations, which would have made the task extremely straightforward.

Instead, I opted to utilise webhooks, a mechanism in web development that enables real-time communication between different applications. In essence, webhooks are HTTP call-backs that allow one system to send real-time data to another as soon as a specific event occurs. This event-driven architecture is instrumental in facilitating automation.

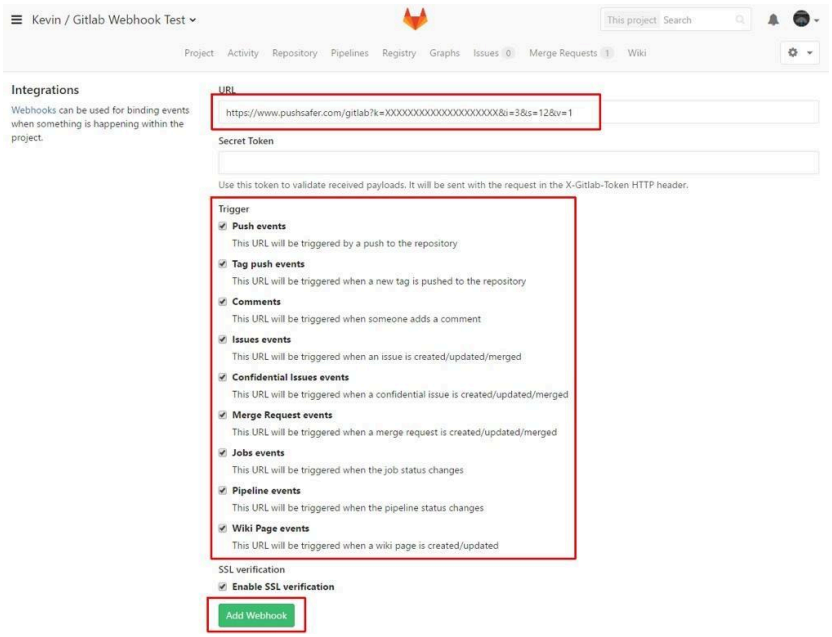

Figure 1: GitLab Webhook


```
{
  "object_kind": "push",
  "before": "95790bf891e76fee5e1747ab589903a6a1f80f22",
  "after": "da1560886d4f094c3e6c9ef40349f7d38b5d27d7",
  "ref": "refs/heads/master",
  "checkout_sha": "da1560886d4f094c3e6c9ef40349f7d38b5d27d7",
  "user_id": 4,
  "user_name": "John Smith",
  "user_username": "jsmith",
  "user_email": "john@example.com",
  "user_avatar": "https://s.gravatar.com/avatar/d4c74594d841139328695756648b6bd6?s=8://s.gravatar.com/avatar/d4c74594d
  "project_id": 15,
  "project":{
    "id": 15,
    "name":"Diaspora",
    "description":"",
    "web_url":"http://example.com/mike/diaspora",
    "avatar_url":null,
    "git_ssh_url":"git@example.com:mike/diaspora.git",
    "git_http_url":"http://example.com/mike/diaspora.git",
    "namespace":"Mike",
    "visibility_level":0,
    "path_with_namespace":"mike/diaspora",
    "default_branch":"master",
    "homepage":"http://example.com/mike/diaspora",
    "url":"git@example.com:mike/diaspora.git",
    "ssh_url":"git@example.com:mike/diaspora.git",
    "http_url":"http://example.com/mike/diaspora.git"
  },
  "repository":{
    "name": "Diaspora",
    "url": "git@example.com:mike/diaspora.git",
    "description": "",
    "homepage": "http://example.com/mike/diaspora",
    "git_http_url":"http://example.com/mike/diaspora.git",
```


Figure 2: GitLab Webhook Payload

Figure 1 is a depiction of how webhook are enabled on the GitLab interface. For our purposes, we set “Merge Request events” as the only trigger. Figure 2 is an example of the payload that is sent to Jira whenever a trigger event occurs.

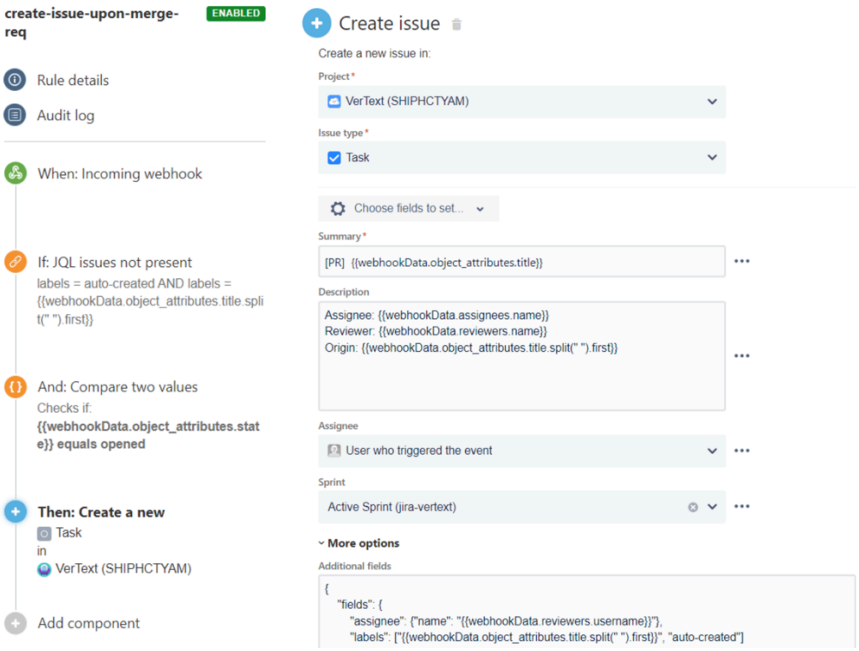

Figure 3: Jira Automation Logic

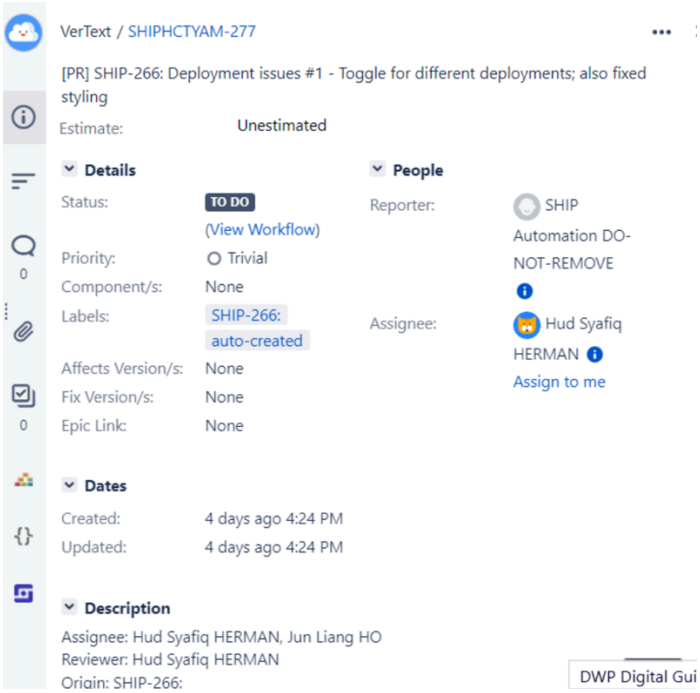

Figure 4: Example of Created Jira Issue

Figure 3 shows how Jira would handle an incoming webhook from GitLab – essentially, if the issue does not already exist, a new issue will be assigned to the reviewer on GitLab. A similar automation was created to automatically set the Jira task as “Done” once the PR has been merged into the main codebase.

It would also be beneficial if all issues, not just PRs, were synchronised between GitLab and Jira – i.e. for Jira issues to appear as a native GitLab issue and vice versa. However, it seems that there is no way to do so without GitLab's Jira integrations. As an alternative, I found that there was a feature on GitLab that allowed for Jira issues to be displayed in a separate tab on GitLab, allowing for easier access - editing Jira issues on Gitlab, however, was not possible.

2.1.2 Update Logo for Document Classification - Frontend

Not all VerText services are on-prem, which could lead to information leaks should a data breach occur. Hence, there is a need to restrict the classification level of the documents used. Previously, the classification level was indicated in the logo image, which was troublesome to update.

Hence, I replaced the logo image (with classification level) with a clean one (without classification level). I added an environment variable that indicates the classification available, and displayed it on the web-app as text.

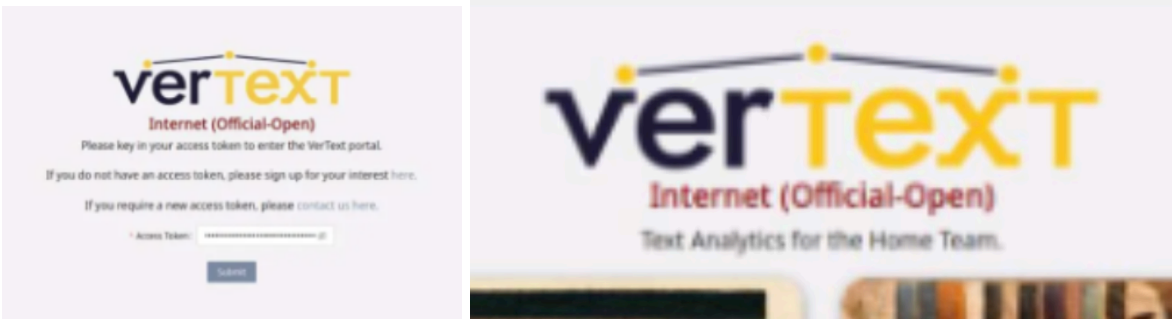

Figure 5: New Logo with Classification Level

Figure 5 shows the updated logo with classification level as text. After this change, the classification level could be easily edited via the environment variable, without a need to create a new logo every time.

2.1.3 Revamp Landing Page Design - Frontend

The original landing page was clunky and hard to use as users had to scroll quite a distance down the page, giving their fingers a good workout, before accessing the services they were interested in. Additionally, the buttons to access the services were tiny and hard to see. Hence, I was tasked with redesigning the landing page.

Figure 6: Original Landing Page

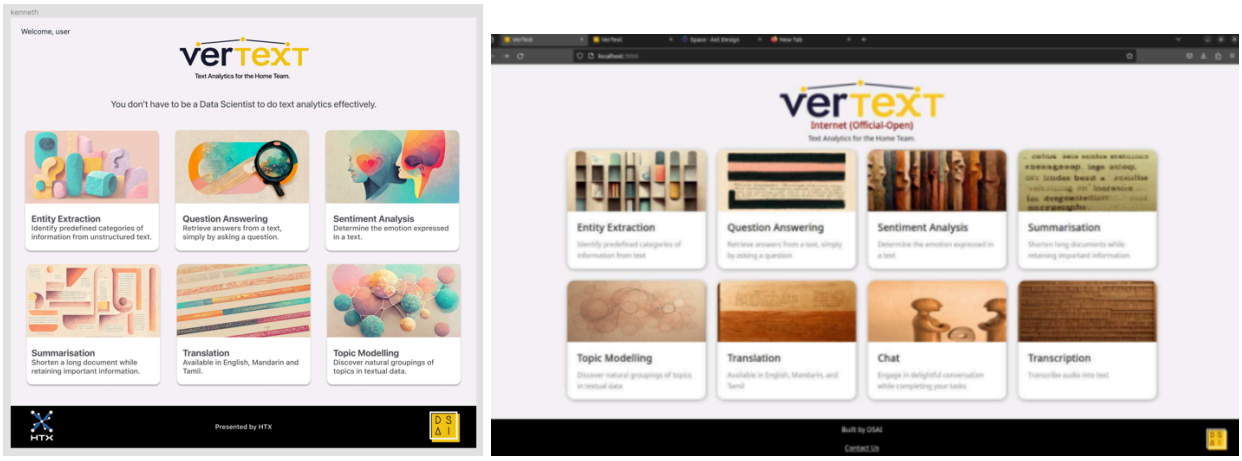

Figure 7: Redesigned Landing Page (Left: Figma, Right: Implementation)

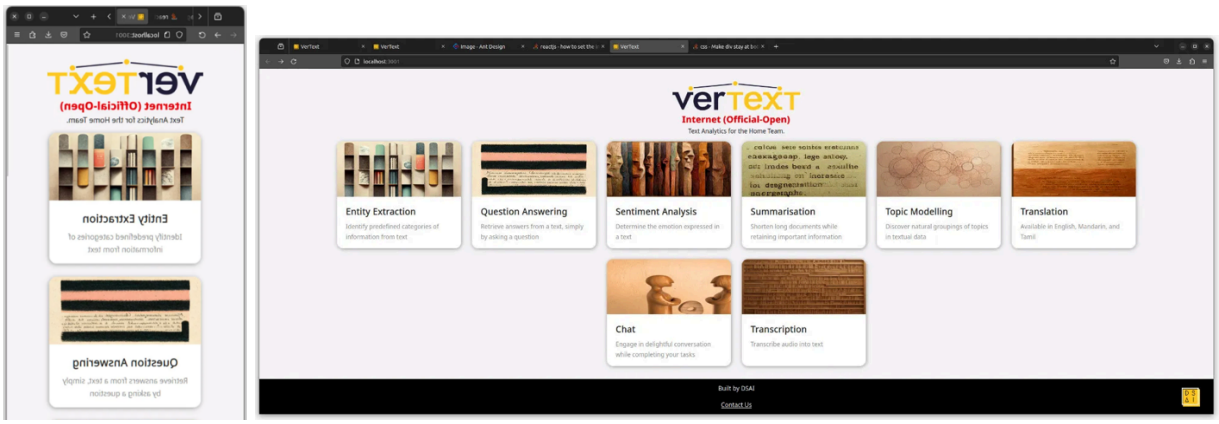

Figure 8: Responsive Landing Page Design (Mobile and Ultrawide)

Figure 6 showcases the original landing page, while Figure 7 depicts the Figma redesign and implementation. The redesign was created with responsiveness in mind, and would adjust what was shown depending on the width of the screen as seen in Figure 8, allowing for comfortable use on devices with different sizes.

2.1.4 Design and Create New Login Page for SSO - Frontend

One of the future features being considered was Single Sign-On (SSO) for Microsoft Entra ID (formerly Azure Active Directory). SSO is an authentication method that enables users to securely authenticate with multiple applications with only one set of credentials.

The original login page only allowed for logging in via an access token. Hence, there was a need to redesign the login page to allow for an alternative login method.

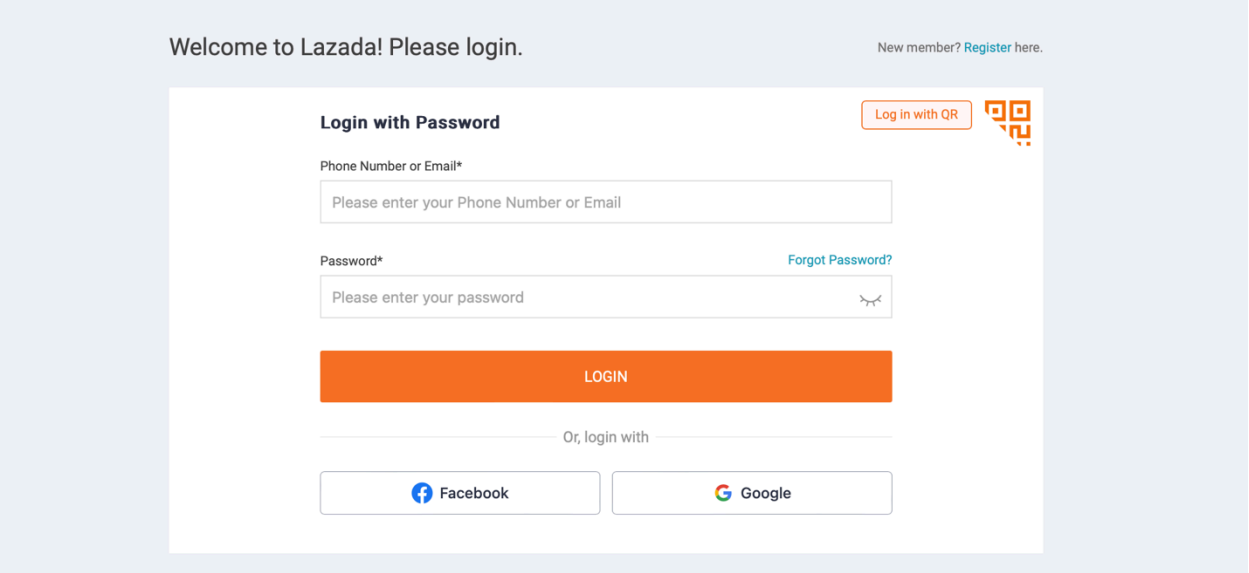

Figure 9: Lazada Login Screen

The inspiration for the design came from Lazada's login screen as seen in Figure 9, which allowed for SSO with Facebook and Google.

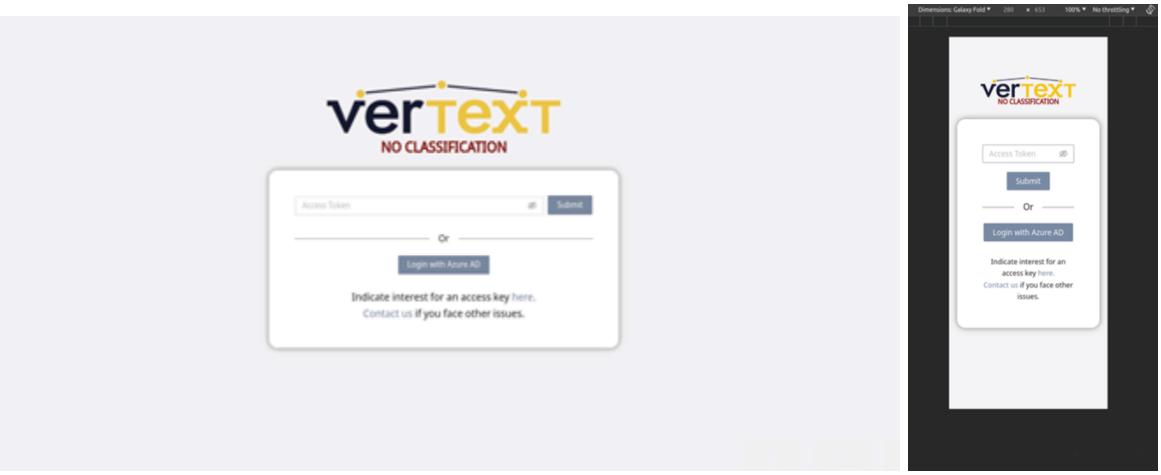

Figure 10: Revamped Login Screen

Figure 10 depicts the redesigned login screen. Once again, responsiveness was an important consideration, which ensures a smooth experience for the user, who might have small hands and hence use smaller devices.

2.1.5 Refactor Frontend Codebase for Reusability and Consistency - Frontend

Each developer has their own coding style and preferences. As numerous developers worked on VerText's many features, the codebase was rather messy, especially since there was no predefined best practice for

them to follow. Additionally, as the developers worked in silos for each feature, they typically recreated components, such as containers, text boxes, and buttons. This led to inconsistent design across the web-app, with slight variations across different pages.

Clean and consistent code is paramount in software development for several reasons. Firstly, clean code enhances readability, allowing developers to easily understand and navigate through the logic of the program. This facilitates debugging and onboarding of new team members. Consistency in coding style and structure ensures uniformity and predictability in the codebase, fostering maintainability. Most importantly, it contributes to bug prevention, especially in styling, as clear and uniform code is less prone to errors and unintended consequences.

Hence, there was a need to refactor the frontend codebase to ensure that the code was clean and consistent.

Figure 11: InputText Component

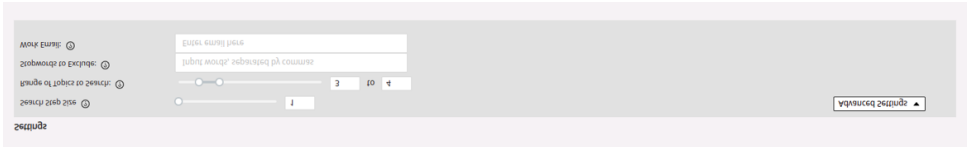

Figure 12: SettingsBox Component

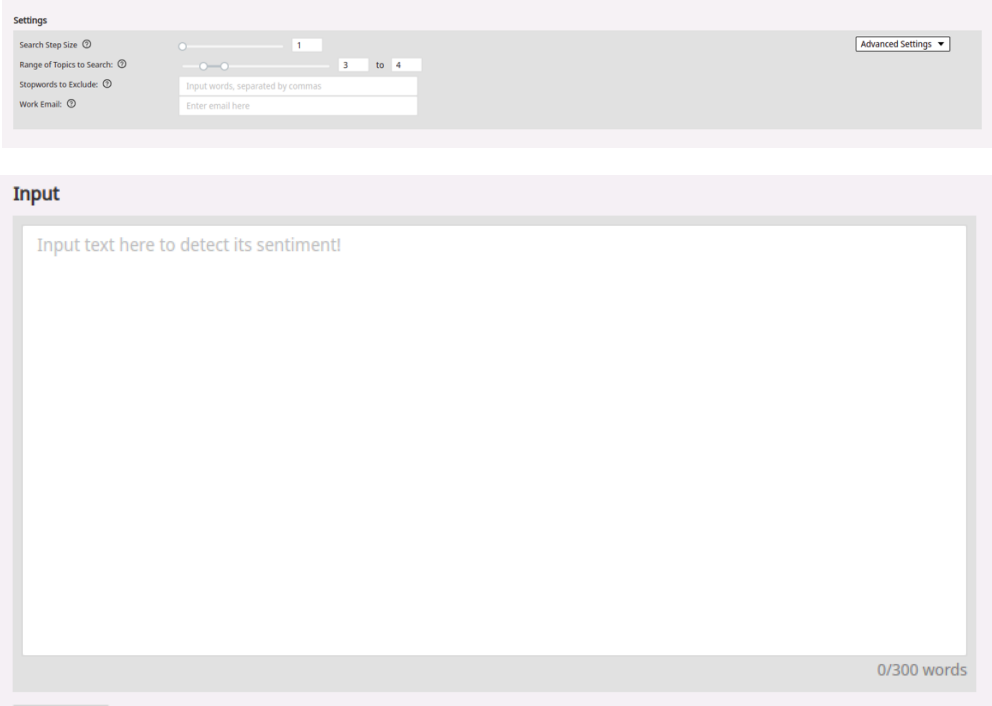

Figure 13: ContainerCard Component (Surrounding Container)

The decision was to create reusable components, InputText, SettingsBox, and ContainerCard, as shown in Figures 11, 12, and 13 respectively. The InputText component would have a fixed width and height, with variable word count (minimum and maximum) and placeholder text. Additionally, an error message should be displayed when the word count conditions were not met. The SettingsBox component would utilise antd's grid to distribute settings in three columns, while advanced settings would be displayed in its own box. Additionally, the advanced settings button would be an indicator of whether advanced settings was active. The ContainerCard component would be a grey background to house other components such as InputText and SettingsBox. It would implement fixed padding and have a fixed size that can be overwritten.

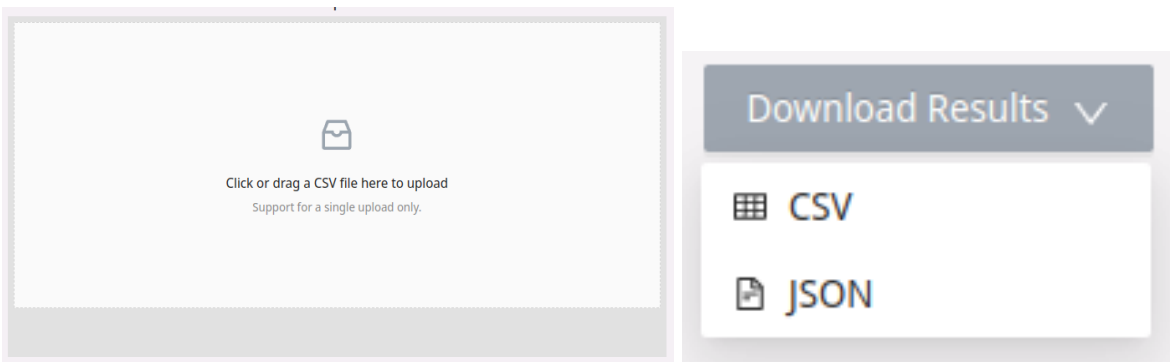

Figure 14: UploadDrag and MultiFormatDownloadButton Components

Additionally, the components UploadDrag and MultiFormatDownloadButton, as shown in Figure 13, which were already created, would also be centralised and reused across pages.

```
return (
  <div>
    <div className={commonStyles.componentContainer}>
      <MainDescription
        title={"Question Answering"}
        subTitle="Extracts an answer from a text input when given a question."
      />
      <div>
        <QASettings
          advancedSelected={advancedSelected}…
          resetParams={resetParams}
        />
      </div>
      <div>
        <Row gutter={50}>
          <Col xs={24} md={12} lg={12}>
            <div className={commonStyles.sectionTitle}>Input</div>
            <ContainerCard>…
            </ContainerCard>
            <SubmitButton handleClick={handleClick} isLoading={isLoading} disableSubmit={disableSubmit} />
          </Col>
          <Col xs={24} md={12} lg={12}>
            {/* this section was not factored out as it is not used on any other pages */}
            <div className={commonStyles.sectionTitle}>Results</div>
            <ContainerCard>…
            </ContainerCard>
            <MultiFormatDownloadButton
              downloadAvailable={downloadAvailable}…
              id={"downloadButton"}
            />
          </Col>
        </Row>
      </div>
    </div>
    <div className={commonStyles.errorModalContainer}>
      <ErrorModal resState={resState} />
    </div>
  </div>
);
```

Figure 15: New Overall Coding Structure

Figure 15 shows the updated code structure that would be consistent for each page. There are 4 sections in total, for the MainDescription, Settings, Body, and ErrorModal components. Previously, the css display of the Body component was not consistent – some pages used flex while others used grid. The decision was to stick with grid for all pages as it allowed for easier implementation of responsive design – with the antd components, Row and Col, the Input and Results box, which are usually displayed in a row, would be displayed in a column should the screen width be small.

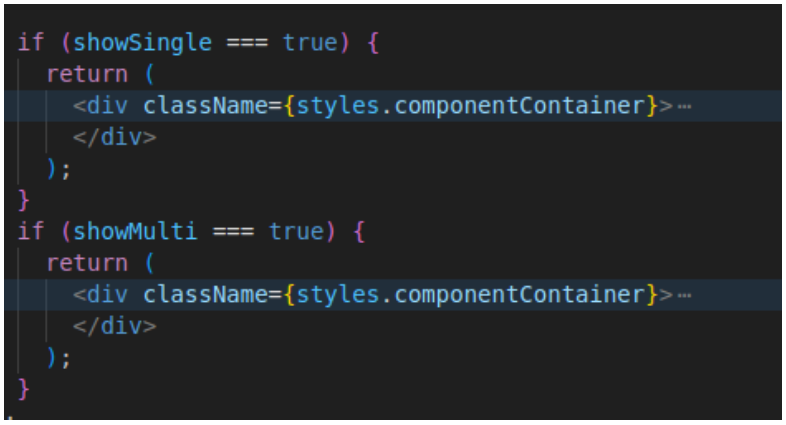

Figure 16: Previous Single / Multi State Logic

Another identified issue was the way single / multi input state was handled in the original code. Figure 16 depicts the original implementation, which returns an entire page depending on the state. This is inefficient as the only differences are within the Body of both return statements while the other components, MainDescription, Settings, and ErrorModal are duplicated. Hence, changes to the page would have to be made twice, which can lead to irregularities should mistakes occur.

Figure 17: New Single / Multi State Logic

Figure 17 shows the new way the single / multi input state is to be handled. Using an inline if statement is more concise and easier to understand. Additionally, there is no longer a need to repeat the above-mentioned shared components.

```css
.errorModalContainer {
  position: fixed;
  right: 1.5%;
  bottom: 2%;
  padding-top: 20px;
}

.componentContainer {
  padding-left: 1.5%;
  padding-right: 1.5%;
  display: flex;
  flex-direction: column;
  justify-content: flex-start;
  background-color: #f5f3f5;
  gap: 12px;
}

.sectionTitle {
  font-weight: bold;
  font-size: 15px;
  margin-bottom: 5px;
  margin-top: 5px;
  color: #333;
}
```

Figure 18: Shared Styles in *common.module.css*

The old code also had individual *index.module.css* files for each page. This is inefficient as changes to page styling will have to be propagated across all pages else there will be inconsistencies. Hence, I used the *common.modules.css* file as shown in Figure 18, which is imported as *commonStyles* in every page, to store the three styles used in all pages – *errorModalContainer*, *componentContainer*, and *sectionTitle*. This negated the need for an *index.module.css file* for every page. Now, *index.module.css* files are only used for unique components, most of which are centralized and reusable.

Figure 19: Example of Irregular Alignment of Text and Buttons

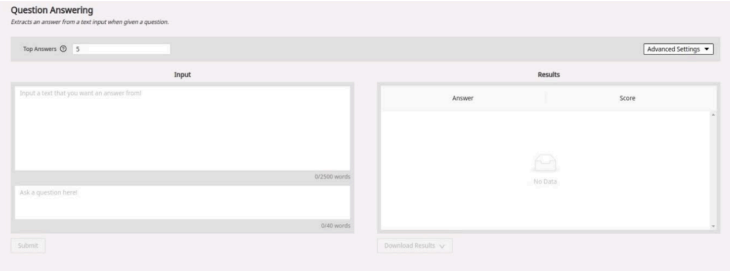

Figure 20: Example of Consistent Alignment

There were also other inconsistencies across the pages that were addressed. For example, as seen in Figure 19, the text and buttons on some pages were aligned irregularly. This issue mostly took care of itself when we switched to using the same components for each page, as shown in Figure 20. However, in the end, there were still some irregularities on some pages that would need to be dealt with individually.

2.1.6 Miscellaneous Bug Fixes - Frontend

There were also some bugs that had to be addressed. For example, the “Not Working” button, which was positioned absolute to the page, blocked buttons positioned behind it when the user scrolled down the page. This issue was not present on every page, however, as some developers added padding to their own pages to mitigate the issue.

```
if (loginSuccess === "true" || REACT_APP_TOKEN_VAL_MODE === "false") {
  return (
    <Router>
      <Switch>
        <Route exact path="/" component={HomePage} />
        <Layout>
          <LayoutNav
            isCollapsed={isCollapsed}
            setIsCollapsed={setIsCollapsed}
            isBrokenBreakpoint={isBrokenBreakpoint}
            setIsBrokenBreakpoint={setIsBrokenBreakpoint}
            content={appcontent}
          />
          <Layout style={{ marginLeft: isBrokenBreakpoint ? 0 : 200 }}>
            <LayoutHeader content={appcontent} />
            <LayoutContent content={appcontent} />
            <div className="notWorkingPadding" />
          </Layout>
        </Layout>
      </Switch>
    </Router>
  );
```

Figure 21: Added Padding in Entry Point

Hence, there was a need to ensure that there was padding on every page, new or old. I chose to add padding to the central layout file instead of solely adding padding to the pages on which the problem occurred. This way, the required padding would be applied to newly created pages as well, ensuring that newly onboarded developers do not get confused about where the padding was applied, or add inconsistent padding to their pages. Additionally, I removed the padding added to individual pages to ensure that the styling was consistent across pages.

During my time in HTX, Vulnerability Assessment and Penetration Testing (VAPT) was conducted. I was assigned to mediate one of the issues, “Unsafe Use of Target Blank”, which was due to the *target="_blank"* attribute on the *CSVLink* component. This was a problem because the site is exposed to performance and security issues:

1. The other page may run on the same process as the original page. If the other page is running a lot of JavaScript, your page's performance may suffer.
2. The other page can access the original page's window object with the window.opener property. This may allow the other page to redirect your page to a malicious URL.

The suggested solution was to simply add the *rel="noopener noreferrer"* attributes, which prevent external links from taking control of the originating browser window and keeps them from knowing that there are links to their material on the site.

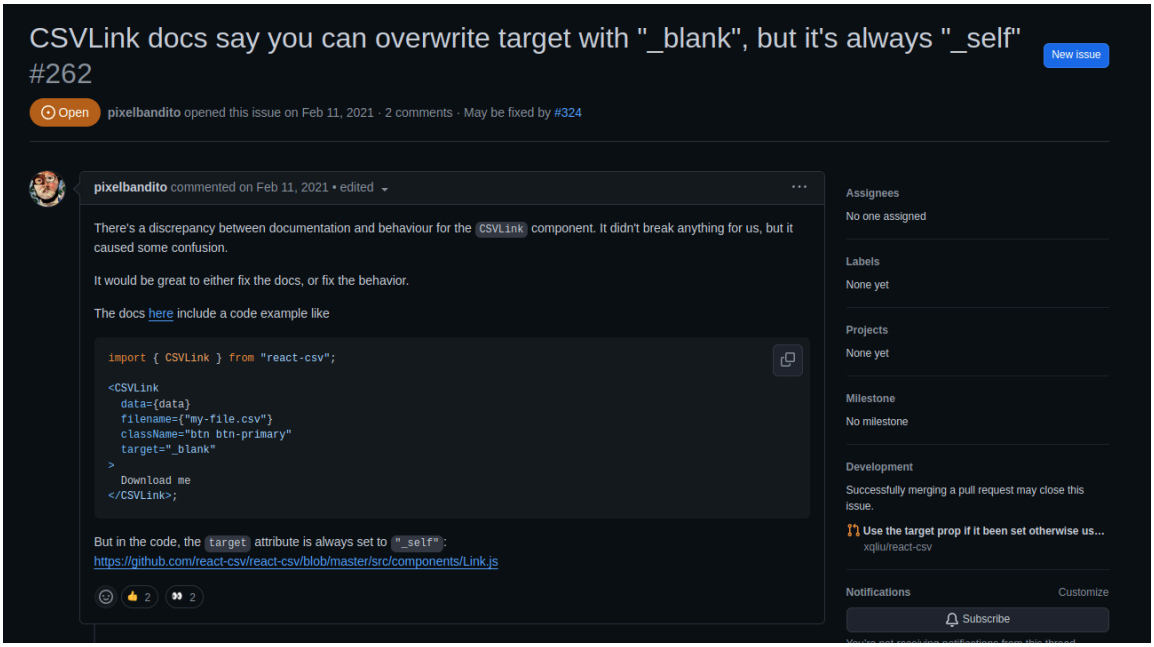

Figure 22: CSVLink Component Attribute Issue

However, upon further investigation, it was discovered that the *target="_blank"* attribute on the *CSVLink* component attribute was buggy - as seen in Figure 22, adding the *target="_blank"* attribute did not affect the behavior of the *CSVLink* component at all. Hence, a better solution to the VAPT issue was to simply remove the *target="_blank"* attribute instead, as there would be no changes in behaviour.

3 Generative Artificial Intelligence (GenAI)

Generative Artificial Intelligence is a subset of AI that focuses on creating new content or information. It utilizes generative models, such as Generative Adversarial Networks (GANs) or autoregressive models, to generate data that is similar to, but not identical to, the existing dataset it was trained on.

In the GenAI team, I focused on exploring ways to integrate knowledge graphs and Large Language Models (LLMs) to improve multi-hop QA performance. Multi-hop QA involves complex queries that cannot be answered directly but require a series of sequential steps to gather and integrate information from various sources - which is difficult for typical RAG with vector databases to perform well in.

My end product was successfully productionised as a Python script that can be easily modified and used by others.

3.1 Description of Work

My time in GenAI introduced me to various aspects of data science, from data scraping to using on-prem Large Language Models. I took on the research process, reading up on relevant articles and testing out their methods through trial and error to determine which was the most performant.

3.1.1 Cleaning Up Judiciary Data

To create knowledge graphs, I first needed data. It was decided that I would use judgements and case summaries from SG Courts which a previous intern had scraped from the site. Unfortunately, he did not take into account that a single case could have multiple categories, and the dataset only contained one category per case.

Hence, my first task was to adjust the scraping script, which utilised the Python package BeautifulSoup, to scrape all relevant categories.

```
category = case.find('a', class_='gd-cw').text.strip()\n",
# category = case.find('a', class_='gd-cw').text.strip()\n",
category_list = case.find_all('a', class_='gd-cw')\n",
category = []\n",
for cat in category_list:\n",
    category.append(cat.text.strip())\n",

title = case.find('a', class_='gd-heardertext').text.strip()\n",
```

Figure 23: Fixed Category Scrapping

As seen in Figure 23, this was quite simple as all I had to do was use *find_all()* instead of *find()*.

Unfortunately, the script was disconnected after a few hours. I realised that this could be because the script was set to have a constant delay between requests, which might have led to the program being detected as a bot by the site. Hence, I increased the delay slightly, and used *randint()* to vary the delay times. This solution worked, and I managed to scrape over 9000 judgements from the site in a couple of hours.

Next, I had to clean the data. I decided on extracting the following data: *legal_case_text*, *categories*, *title*, *case_id*, *decision_date*, *legal_citation*, *case_year*, and *case_type* (which represented the court the case was held in: Family Justice Court, Supreme Court, etc). My first attempt was to iterate over each row to extract and clean the necessary information. However, as there were over 9000 lines, it took quite a significant amount of time. Hence, I decided to look into alternative methods to speed up the process - I found that by storing the data as a Pandas DataFrame, I could use the *apply()* function, which greatly sped up the program runtime. Upon further research, this is because the Pandas library uses numpy, which implements much more efficient array operations.

Figure 23: Fixed Category Scrapping

As I initially expected some runtime, I decided to use the tqdm library alongside Pandas to see the progress. However, upon running the code, I realised that it was not needed, as it barely took a second to clean up a column, as seen in Figure 24.

The final result was about 500mb worth of text, which was quite difficult to handle. For practical purposes, I decided to sort the data by the case type (type of court). There are 8 different courts in total.

3.1.2 Knowledge Graphs with OpenAI Functions

Source: [Constructing Knowledge Graphs from Text using OpenAI Functions]

With the advent of LLMs such as OpenAI's GPT-3.5, extracting structured information from unstructured text has become much easier. In this article, the author discusses the construction of knowledge graphs using OpenAI functions and LangChain, with a focus on Neo4j integration. The information extraction pipeline is as such: Conference Resolution -> Named Entity Recognition (NER) -> Entity Disambiguation -> Relationship Identification.

```
    prompt = ChatPromptTemplate.from_messages(
        [(
          "system",
          f"""# Knowledge Graph Instructions for GPT-4
## 1. Overview
You are a top-tier algorithm designed for extracting information in structured formats to buil
- **Nodes** represent entities and concepts. They're akin to Wikipedia nodes.
- The aim is to achieve simplicity and clarity in the knowledge graph, making it accessible fo
## 2. Labeling Nodes
- **Consistency**: Ensure you use basic or elementary types for node labels.
  - For example, when you identify an entity representing a person, always label it as **"pers
- **Node IDs**: Never utilize integers as node IDs. Node IDs should be names or human-readable
{'- **Allowed Node Labels:**' + ", ".join(allowed_nodes) if allowed_nodes else ""}
{'- **Allowed Relationship Types**:' + ", ".join(allowed_rels) if allowed_rels else ""}
## 3. Handling Numerical Data and Dates
- Numerical data, like age or other related information, should be incorporated as attributes
- **No Separate Nodes for Dates/Numbers**: Do not create separate nodes for dates or numerical
- **Property Format**: Properties must be in a key-value format.
- **Quotation Marks**: Never use escaped single or double quotes within property values.
- **Naming Convention**: Use camelCase for property keys, e.g., `birthDate`.
## 4. Coreference Resolution
- **Maintain Entity Consistency**: When extracting entities, it's vital to ensure consistency.
If an entity, such as "John Doe", is mentioned multiple times in the text but is referred to b
always use the most complete identifier for that entity throughout the knowledge graph. In thi
Remember, the knowledge graph should be coherent and easily understandable, so maintaining cor
## 5. Strict Compliance
Adhere to the rules strictly. Non-compliance will result in termination.
          """),
            ("human", "Use the given format to extract information from the following input: {
            ("human", "Tip: Make sure to answer in the correct format"),
        ])
```

Figure 24: Prompt Template for Conference Resolution and NER

The author utilised GTP-3.5 for conference resolution and NER, as seen in Figure 24, partly skipping over entity disambiguation, which could lead to nodes that refer to the same entity. Additionally, the author specified the allowed node labels to prevent the formation of similar nodes, such as Company and Organisation. This problem is especially significant with relationship types, which we faced later on.

```
In [12]: cypher_chain.run("When was Walter Elias Disney born?")

> Entering new GraphCypherQAChain chain...
Generated Cypher:
MATCH (p:Person {name: "Walter Elias Disney"}) RETURN p.birthDate
Full Context:
[{'p.birthDate': 'December 5, 1901'}]

> Finished chain.
Out[12]: 'Walter Elias Disney was born on December 5, 1901.'
```

Figure 25: Demonstration of GraphCypherQAChain

The information in the Neo4j knowledge graph can be browsed by constructing Cypher statements (Cypher is a structured query language used to work with graph databases, similar to how SQL is used for relational databases). We used LangChain's GraphCypherQAChain for this purpose - it reads the schema

of the graph (node labels and relationship types) to construct an appropriate Cypher statement based on the user input as demonstrated in Figure 25.

Upon cloning the provided git repository and testing it on the judiciary data, results were abysmal. There were hundreds of node labels and relationship types for just a single judgement. Even upon specifying the allowed node labels and allowed relationship types, the problem remained. Nevertheless, it did not make sense to restrict the allowed relationship types anyways - doing so would have caused the database to lose much meaning and nuance. Hence, our only mitigation was to restrict the allowed node labels to allow for a cleaner graph. This however, proved to be quite a challenge as well.

I started by editing the system prompt to repeatedly emphasise the need to keep to the allowed relationship types. Whilst this was effective for the first few cases, it tends to hallucinate relationship types after going through a few judgements.

As a result, the GraphCypherQAChain function was unable to perform - with too many similar node labels, it would frequently query the wrong node label and fail to retrieve the context. However, this method performed quite well for small amounts of data. Deleting nodes with illegal labels also allowed the GraphCypherQAChain function to better retrieve context from whatever remained - doing so, however, obviously led to the loss of large amounts of data, and this practice was hence not continued.

3.1.3 Knowledge Graphs with GraphGPT

Source: GraphGPT: Graph Instruction Tuning for Large Language Models

The paper introduces the GraphGPT framework, the goal of which is to develop a graph-oriented LLM that exhibits high generalization across diverse downstream datasets and tasks, even in the absence of information from the downstream graph data. GraphGPT achieves this by aligning LLMs with graph structural knowledge through a graph instruction tuning paradigm.

The framework includes a text-graph grounding component, establishing a connection between textual information and graph structures. Additionally, it proposes a dual-stage instruction tuning paradigm, complemented by a lightweight graph-text alignment projector. This paradigm explores self-supervised graph structural signals and task-specific graph instructions, guiding LLMs to understand complex graph structures and enhancing their adaptability across different downstream tasks.

The paper postulates that when GraphGPT is evaluated on both supervised and zero-shot graph learning tasks, it showcases superior generalization compared to state-of-the-art baselines.

Unfortunately, perhaps due to inexperience, I was unable to get the model up and running. After downloading the model's weights, I could not run inference - with the AutoModelForCasualLLM function, I faced the error: KeyError: "GraphLlama", and I faced even more difficulties with the LlamaForCasualLLM function. It seemed that to get this model up and running, I would need to rebuild the model from a Vicuna-7B-v1.5 model (tuning on instruction data and training on Arxiv and Pubmed with Text-Graph grounding). While the instructions were provided on their GitHub repository, I was lacking a modified config.json file, which was required for training.

Upon revisiting their repository while writing this report, I have discovered that as of 261223, the author has updated their training code and provided the missing config file. Perhaps another intern would have better luck with this model!

3.1.4 Knowledge Graphs with Mistral 7B

Source: How to Convert Any Text Into a Graph of Concepts

This article was quite similar to the one featured in 3.1.2 Knowledge Graphs with OpenAI Functions, except for that a local LLM was used instead of OpenAI's GPT, and that another prompt was used. In terms of identifying relationships between entities, the performance was quite similar. The author also included an additional step - calculating contextual proximity by assuming concepts that occur close to each other in the text corpus are related. Additionally, this solution also utilised the NetworkX Python library to group nodes in "communities", which are similar to node labels.

While this approach did not have the issue of runaway labels, the presence of too many relationship types (there were essentially no relationship types, each relationship was expressed as its own edge), made it difficult to form a proper Cypher statement as the list of relationships greatly exceeded the context window of all the LLMs I tried it with (GPT-3.5, GPT-4 and Mistral 7B).

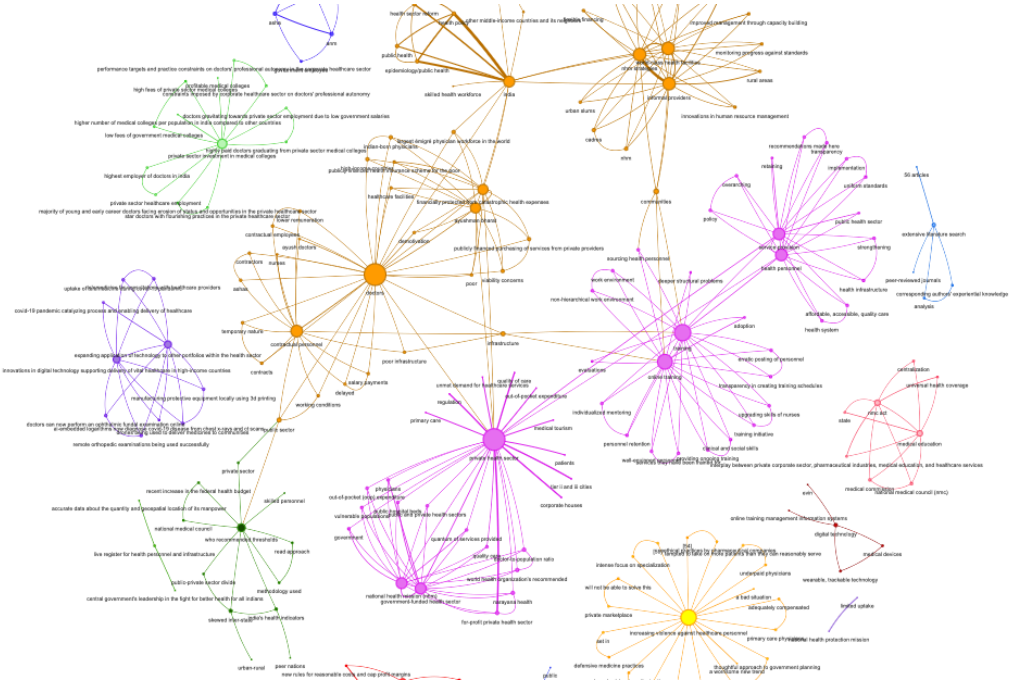

Figure 26: Visualisation with PyVis

When visualised with PyVis, as displayed in Figure 26, the graph was quite nice to look at though. Additionally, I gained more experience in running on-prem LLM models, which was nice.

3.1.5 Improving Responses with "Research Agents"

Source: [The Research Agent: Addressing the Challenge of Answering Questions Based on a Large Text Corpus]

I did not have much luck with knowledge graphs, so I took a temporary hiatus to try out alternative methods to improve the performance of multihop QA.

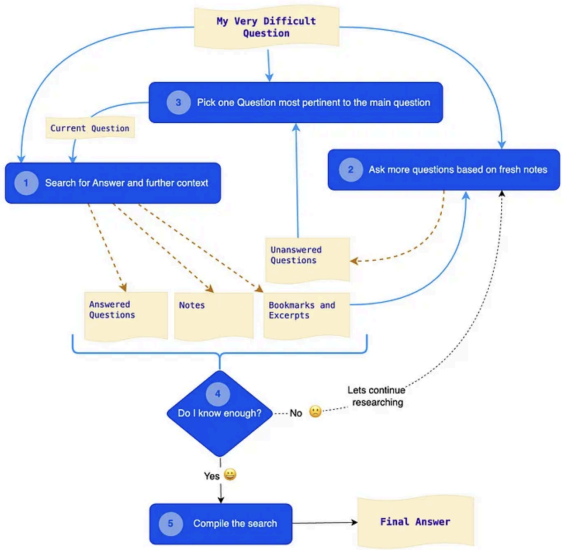

Figure 27: Research Process of the "Research Agent"

In this article, the author explored the concept of a "Research Agent" which follows a typical research methodology:

1. Try to answer the questions with research. In doing so, make a few notes and bookmark excerpts that are relevant.
2. Note down unknowns in the excerpts and write down more questions that allow one to learn more about these unknowns.
3. Choose the question that is most pertinent to the original question, and restart from step 1.

Components

1. **QA Agent —** Search for answers and further context
   This is a simple *'stuff'* Retrieval QA chain that uses a vector store. In the future, this can be an AI Agent that uses tools like vector stores, search APIs or Wikipedia APIs, Moderation APIs and previous research data. The prompt here is tuned to generate succinct answers based on the 1. The context (documents), and 2. the pertinence to the ***original question***.
   Except for the first loop, the ***current question*** is always an intermediate question generated in step 2 and chosen in step 3. The Agent appends the intermediate answer to the notes and the latest excerpts (the documents used to answer the ***current question***) to the bookmarks. The most recent of these documents are utilised in step 2.
2. **Question Generator —** Ask more questions based on fresh notes
   Here, the agent uses the most recent vector search results matching the ***current question*** and uses them to generate more questions pertinent to the ***original question***. It appends these questions to the list of ***unanswered questions***. The prompt here is tuned such that the newly generated questions do not overlap with the existing list of questions.
3. **Most Pertinent Question Picker —** Pick one question most pertinent to the original question
   This prompt picks one question from the list of unanswered questions that is the most pertinent to the ***original question***. This question is used as the ***current question*** for the next loop.
   In the next loop, the agent removes this question from the list of ***unanswered questions*** after generating a fresh set of questions.
4. **Analyser —** Do I know enough?
   I am using a ***max_iterations*** parameter to exit the loop. This works pretty well for now. However, it might be better to dynamically decide on the number of iterations or an exit strategy based on the evolving context. I will work on an 'analyser' that can do this autonomously in the future.
5. **Research Compiler —** Compile the research
   This is the final prompt. It uses the notes made during the research process to arrive at an elaborate `final answer` to the `original questions`.

Figure 28: Components of the "Research Agent"

The author created the "Research Agent" with the components shown in Figure 28. I tested the implementation with the provided vector database and the results seemed promising. As expected, the answers had more detail, and provided additional context relevant to the query. However, the time taken to generate a response increased exponentially.

While this implementation used a vector database instead of a knowledge graph, I believed that the logic could still be implemented if we were to use a different retriever.

Upon revisiting the article while writing this report, I discovered that the author has created a v3 of their implementation, which could potentially have improved performance (I took reference to v2).

3.1.6 Improving Retrieval with Advanced RAG Strategies

Source: [Implementing Advanced Retrieval RAG Strategies With Neo4j]

The author postulated that the traditional approach to vector similarity search could sometimes overlook specific pieces of context when longer text is embedded. By taking advantage of advanced RAG techniques, we can increase the retrieval accuracy while retaining the contextual information.

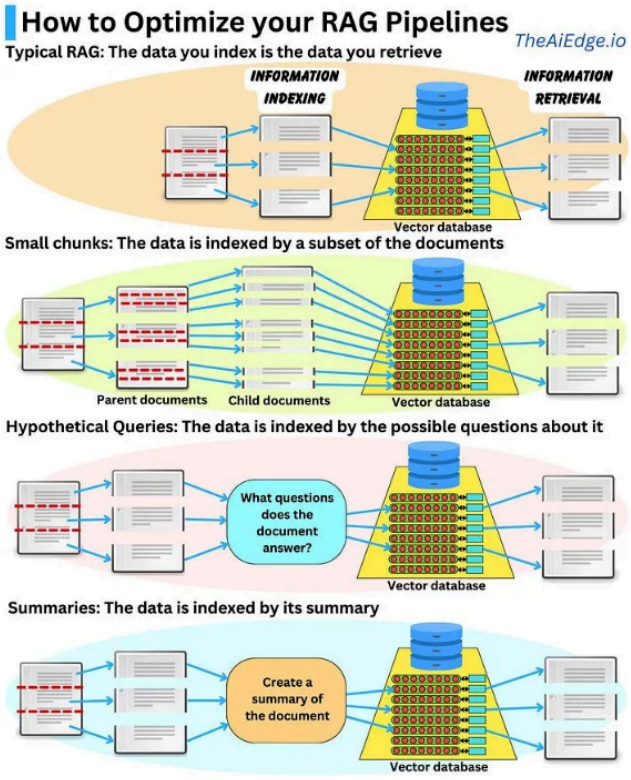

Figure 29: Advanced Strategies

Figure 29 shows the 3 methods provided: Small Chunks, Hypothetical Queries, and Summaries. This turned out to be quite a useful read as I learnt how to implement what was essentially a vectorstore in Neo4j, which was typically used to store knowledge graphs. I also found that implementing the method provided did indeed improve the responses generated.

3.1.7 Combining Graphs and Textual Data

Source: [Knowledge Graphs & LLMs: Multi-Hop Question Answering]

Reading through Neo4j's NaLLM blog, I came across this article, which explained that vector databases had poor performance in multi-hop QA due to three main reasons:

1. Repeated information in top N documents
2. Missing reference information
3. Difficulties in defining ideal N number of retrieved documents

Knowledge graphs could serve as condensed information storage that represents data as nodes and relationships. The knowledge graph representation essentially connects the data and makes it easier to answer questions spanning across multiple documents, leading to better multi-hop QA performance.

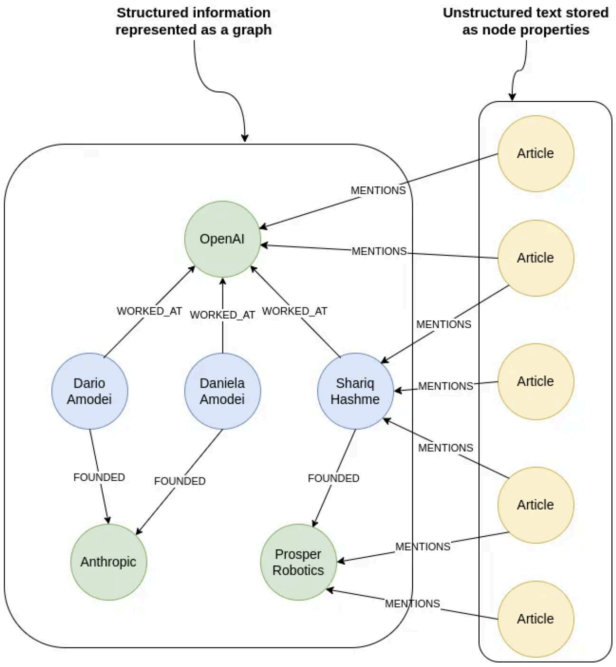

Figure 30: Storing Structured Data Alongside Unstructured Text

Figure 30 was particularly useful as it led to the realisation that there might not be a need to convert each individual judgement into knowledge graphs - I could just store them as unstructured text, and retrieve them according to their attributes such as category or court.

3.1.8 End Product

In the end, the articles I found most influential were those in 3.1.5, 3.1.6, and 3.1.7. While most current research focused on constructing knowledge graphs from unstructured text, those who wanted to easily access the data for RAG purposes would restrict the relationship types, choosing to store only some information deemed necessary across documents, such as age, employment, or subsidiaries. More complex information that might not exist on all documents would then be discarded. Those who manage to represent all their data in the form of a knowledge graph would typically have numerous relationship types, which is not optimal for RAG, but do indeed create nice looking graphs that showcase relationships between entities well.

Essentially, to retrieve information well with Cypher statements, we would need to restrict relationship types. However, in order to store all of our unstructured data as a knowledge graph, we would need to have a great variety of relationship types, owing to the fact that the relationships between entities in court judgements tend to be quite unique across cases.

For our purposes, we had to minimise information loss, and hence could not restrict the relationship types. As a result, it was not feasible to represent all our information as a knowledge graph, since it was pretty much impossible to generate Cypher statements once the knowledge graph exceeded a certain size (2 court judgements turned out to be the limit for me).

Hence, I utilised both vectors and graphs to represent the information in the judiciary hearings. Similar to the architecture shown in Figure 30, I represented all structured data in a graph, but instead of directly linking the relevant text to the nodes, I chose to use what was essentially a vector database in Neo4j (as demonstrated in 3.1.6) to take advantage of advanced RAG strategies.

Adding on to the code provided by the above-mentioned articles, I first created an “ingester” that would take as input the judgements and store chunks as Parent nodes on a Neo4j aura instance, along with their embeddings. For each Parent node, it would generate a Summary node, 10 hypothetical Question nodes and 10 Child nodes. Afterwards, it would create a Detail node for each case consisting of the case metadata, and Category nodes for each unique category, which would be linked to the appropriate Detail node.

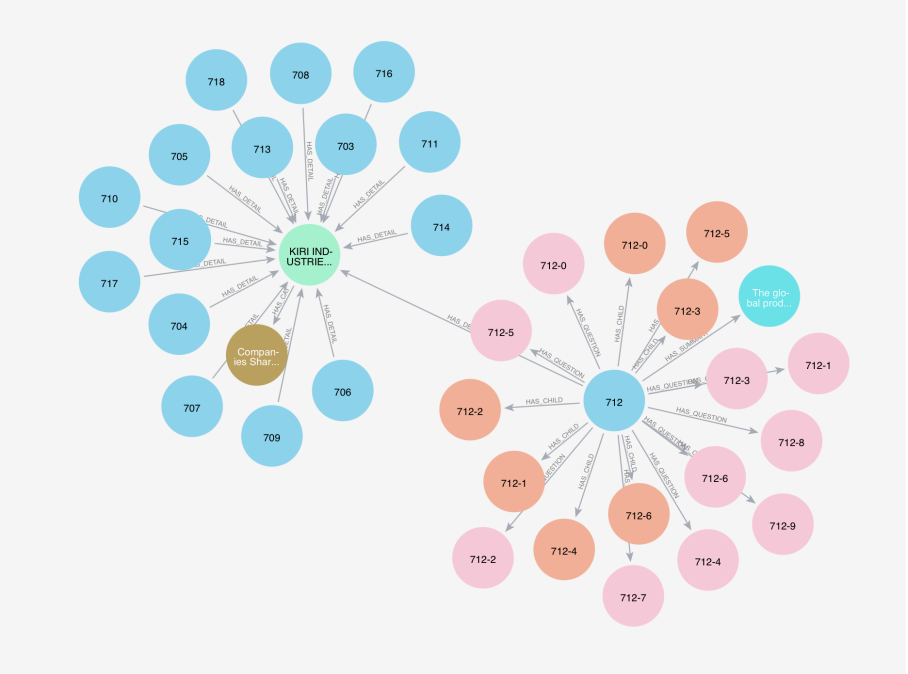

Figure 31: Graphical Visualisation of an Individual Judgement

The product is shown in Figure 31. The Details node (green) has many Parent nodes (blue) associated with it, and each Parent node (blue) has a Summary node (teal), 10 Question nodes (pink) and 10 Child nodes (orange) associated with it.

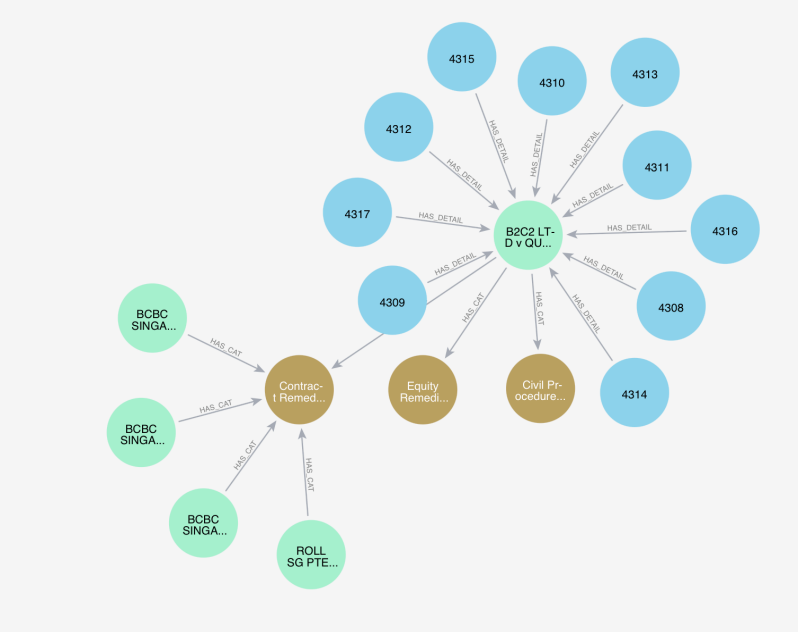

Figure 32: Detail Nodes and Category Nodes

Each Detail node can have multiple Category nodes (brown) associated with it and each Category node (brown) can have multiple Detail nodes associated with it, as shown in Figure 32.

```
def case_digester(case_id, question, method) -> str:
    """
    This function utilises the Neo4j vectorstore to retrieve a parent node.\n
    Options for method include: QUESTION, CHILD, SUMMARY
    """
    hypothetic_question_query = """
    MATCH (node)<-[:HAS_""" + method + """]-(parent: Parent)-[:HAS_DETAIL]->(d:Detail)
    WHERE d.case_id = '""" + case_id + """'
    WITH parent, max(score) AS score // deduplicate parents
    RETURN parent.text AS text, score, {} AS metadata
    """

    print("Searching within case:" + hypothetic_question_query)

    hypothetic_question_vectorstore = Neo4jVector.from_existing_index(
        OpenAIEmbeddings(),
        index_name="hypothetical_questions",
        retrieval_query=hypothetic_question_query,
    )

    template = """Answer the question based only on the following context:
    {context}

    Question: {question}
    """

    prompt = ChatPromptTemplate.from_template(template)

    model = ChatOpenAI()

    retriever = hypothetic_question_vectorstore.as_retriever()
```

Figure 33: Single Case Retrieval

Users can ask a question with regards to a specific case, as seen in Figure 33.

```
def retrive_cases_from_cat(category):
    graph = Neo4jGraph()
    cases = graph.query(
        """
        MATCH (d:Detail)-[:HAS_CAT]->(c:Category)
        WHERE c.title = $cat
        RETURN distinct d;
        """, {"cat": category})
    return [case["d"]["case_id"] for case in cases]
```

Figure 34: Single Category Retrieval

They can also ask a question with regards to a certain category, as in Figure 34, which shows the Cypher statement used to retrieve all relevant cases. The goal was to create a single “Research Agent” that would review all relevant cases and generate a response accordingly. However, due to time constraints, I was unable to meet this goal. Instead, the script would go through the research process for each individual case, before summarising all responses into one.

Additionally, the script could be better improved with an analyser that would determine if the information collected is enough and exit the research loop if so. The current implementation uses a max_iterations parameter to exit the loop, which could sometimes lead to too much, or too little context.

4 Reflections and Acknowledgements

This internship has proven to be an enriching experience. I was provided insight to the role of a front-end engineer via the development of VerText, where I learnt how to use the Agile methodology as a project management approach to simplify the development process by breaking down the project into phases with continuous collaboration and improvement. Looking ahead, I aspire to broaden my skills by exploring backend development to become a more proficient full-stack engineer.

My most enjoyable experience, however, was in the GenAI team, where I worked on nascent research on integrating knowledge graphs and LLMs. I was given a thorough initiation into the research process, and while many of the created scripts did not work as well as intended, I learnt a wide variety of skills and got exposure to the different aspects of data engineering through team sharings.

In closing, I would like to extend my thanks to Shisheng for the opportunity to intern at xData and for taking the care to assign me projects closely aligned with my interests. Much thanks to my supervisor, Wei Jie, who dedicated time during our weekly check-ins to provide support and for helping out whenever I got stuck. Special thanks to Jason, who spent much time guiding my journey in GenAI and explaining concepts I found hard to understand. The past 2 months I have spent at HTX have definitely exceeded my expectations, and I am extremely grateful for the invaluable opportunity.