Department of Computer Science
Individual Project – CS3IP16

Implementing a platform to employ modern machine learning and data mining techniques to identify instances of plagiarism and fraudulent behaviour in published works

Ben Magee
vv008146

Supervisor: Dr. Varun Ojha

27th April 2019

1 Abstract

The report following details the technical implementation and challenges of a plagiarism detection platform using only lexical features and document structure as sources of information. It describes the architecture of the platform, as well as successes and challenges in detecting plagiarism in this manner.

Overall, the platform has been implemented using a microservices architecture, allowing for horizontal scaling in cases of high demand, self-healing if a service or physical host fails, and extensibility due to the decoupled nature of the services that comprise the platform. This will make extending the platform later to add additional test services very easy.

Work remains to be done to improve the accuracy of plagiarism detection using solely lexical features. Promising results have been shown in the areas of language classification and homoglyph detection, while results are considerably more varied when analysing lexical features like punctuation use and sentence and word length.

2 Acknowledgements

Thank you to Dr. Varun Ojha, not only for his support during the implementation of this project, but also for his dedication and hard work as a module convenor for this course.

3 Table of Contents

4 Glossary

Term	Definition
OIDC	OpenID Connect [1] standard. An open-source standard that defines an authentication and authorization framework, built upon the OAuth2 specification. Supports authentication and authorization from a range of client platforms.
OAuth2	An authorization specification [2]. Allows users to grant applications access to their data, and to share information with third-party websites.
API	An interface exposed by an application that allows third parties to interact with that service in a well-defined manner.
Docker	A container technology which allows an application and all of its dependencies to be packaged into a single unit, which can then be run anywhere, regardless of the software on the operating system.
Kubernetes	A container orchestration platform. Kubernetes schedules workloads on hardware hosts. If a host fails, the workload is rescheduled elsewhere. It also handles scaling and communications.
RabbitMQ	A message queue broker. RabbitMQ delivers messages from producers to consumers in a highly configurable way. A RabbitMQ broker can either be a single instance, or configured in a highly-available manner.
SVM	Support Vector Machine
TFIDF	Term frequency inverse document frequency – a method of identifying keywords in a document
Sharding	Storing multiple copies of data across several physical hosts to protect against data loss

5 Introduction

This report aims to detail the efforts of implementing a plagiarism detection platform which bases its findings on the lexical elements of the submitted document, in combination with previous submissions by the same author.

The final product should conform to the following objectives:

- The platform should be able to detect instances of plagiarism based on lexical features alone
- The platform should not rely on external datasources
- The platform should allow a user to experience the full lifecycle of document upload to selecting documents to view, to viewing test results

The aim of the platform and its implementation is to determine whether it is possible to design and build a user-friendly plagiarism detection tool, that can effectively detect instances of plagiarism without relying on large databases of commercial texts.

It is my intention to implement this platform in a scalable and well-designed manner using a microservices architecture, so that in the future it may be possible to build upon the work undertaken during the course of this project.

This report will detail research around existing plagiarism detection platforms, investigation into existing research around novel ways of detecting plagiarism, as well as details of the implementation undertaken, including novel approaches, problems encountered, and considerations made in hindsight.

6 Problem Articulation & Technical Specification

Traditional plagiarism detection methods revolve around keyword and phrase matching. Typically, a plagiarism detection platform provider has a large database of previously submitted documents, as well as a corpus of commercially and academically published texts to compare submissions to. This approach is classified as external plagiarism detection [3].

While these systems have an excellent detection rate of content plagiarised verbatim from these sources, they begin to falter under a variety of common, easy to construct conditions. If content is plagiarised from a more obscure source that hasn't been ingested into the platform's database of existing documents, then it is unlikely to be detected. Similarly, small changes in wording or punctuation can often defeat these platforms, and typographical tricks can lead to a computer reading in one phrase, while another is displayed to a human reader: this is typically achieved using homoglyphs or white font [4].

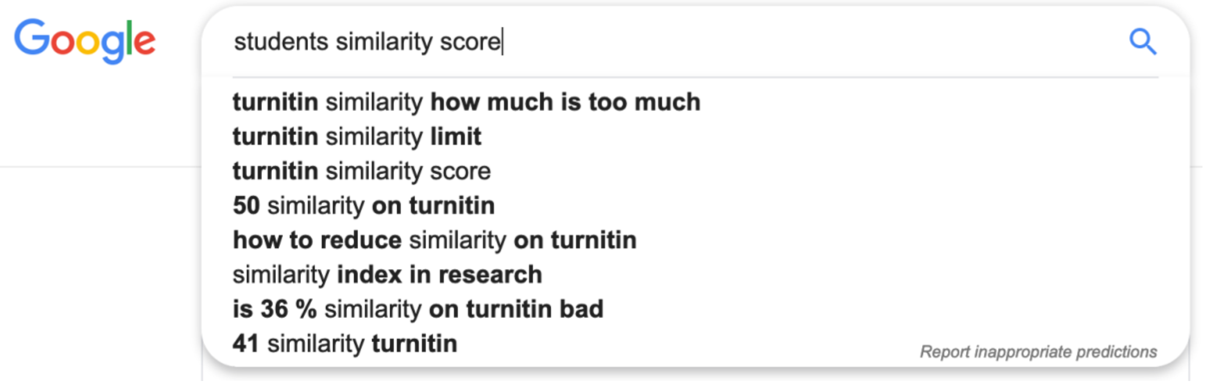

Google
students similarity score
turnitin similarity how much is too much
turnitin similarity limit
turnitin similarity score
50 similarity on turnitin
how to reduce similarity on turnitin
similarity index in research
is 36 % similarity on turnitin bad
41 similarity turnitin
Report inappropriate predictions

Figure 1 Google search predictions displaying confusion over TurnItIn similarity scores

All platforms that take this keyword or phrase matching approach suffer from a common flaw: correctly cited material can be flagged as unoriginal. This happens when an author correctly cites another author's work, but the plagiarism detection platform detects the use of material it has indexed into its database. This makes the "similarity score" that many of these services generate difficult to interpret: a body of work may have a comparatively high similarity score simply because it frequently cites pre-existing work, and does so in a completely legitimate manner. Confusion around what a generated similarity score represents may deter students from referencing existing text when appropriate, for fear of being penalised by a higher similarity score – figure 1 shows evidence of this.

Finally, these phrase detection systems don't account for when a work is produced that correctly cites any pre-existing work, and is otherwise academically sound, apart from disingenuity on the submitter's part around the original author of the submitted document. Essay-factory websites continue to become more popular and widespread, with little achieved by the academic sector to tackle their increasing pervasiveness. Indeed, since the issue was highlighted in 2012 by the BBC [5], the practice has become more prolific, with 2018 seeing advertisements for such services appearing on popular websites like YouTube [6].

The author proposes a more fundamental approach to plagiarism detection that focuses on the building blocks of language, rather than the specific phrasing used by an author in their document. The author posits that just as a person has their own unique mannerisms when speaking, or carrying out other day-to-day tasks, their style of writing is sufficiently distinguished as to be able to ‘fingerprint’ it, and use this model to determine how statistically likely it is that another piece of work, purportedly by the same author, is in-fact original and the author verifiable.

Indeed, a more ad-hoc approach to the same methodology has been acknowledged and practised by educators worldwide, albeit typically of smaller class sizes. Before the advent of the widespread availability of services such as TurnItIn, markers would become suspicious about uncharacteristic phrasing, or use of advanced grammatical structures that they didn’t typically associate with the student and use search engines in order to try and identify the original source of the document or passage.

This innate sense of authorship is rooted in the marker’s understanding of a language’s syntactic structures and their use by a student, and while it is possible to use this sense with a relatively small group of students with whom the marker frequently interacts, it becomes much less accurate as cohort sizes increase (for example, a university course could have a cohort size in the hundreds), resulting in markers becoming less, or not at all, familiar with the student and their previous work. This situation becomes ever more untenable where marking is split between a group of teachers, lecturers, or assistants, as it is even less likely that a marker will be familiar with a student’s normal writing style.

By analysing use of language and structures, it may be possible to quantify this innate perception of authorship, making it possible to scale to large cohorts of students, and providing a marker with empirical evidence of anomalies in a submission. It’s the author’s intention to investigate whether this analysis of language use could be undertaken on both a document-by-document basis, as well with significantly smaller corpuses in order to detect anomalies within an individual document (for example: a suspicious paragraph or phrase within a document).

In order to achieve this, several features of language that are indicative of authorship need to be identified. One such feature would be the frequency of different n-grams in the author’s work. N-grams are n-character slices of longer strings [7]. For example, a common bi-gram in the English language would be “th” or “st”. It is posited that different authors will have different frequency distributions of n-grams in the English language.

Simpler features could also be analysed, average sentence length, word length, frequency and use of different punctuation – all these features play a part in building an author’s unique writing style, and all are quantifiable. Term frequency-inverse document frequency statistics could be used to attempt to identify words that are more commonly used by the author than the general population.

Such analyses of basic language structures can also be used to combat a submitter’s attempts to fool a plagiarism detection system into misreading a phrase. Researchers at the University of Wollongong describe using the index of coincidence of the English language and chi-square tests in order to detect language spoofing in a document [8]. While it is impossible to flag an instance of plagiarism based solely on this statistical metric, the phrase or paragraph can be flagged as not conforming to linguistic norms, and consequently a human marker can examine the passage in more detail.

The desired platform is one that should benefit all stakeholders in the current situation. Academic staff who currently have doubts about the effectiveness of current plagiarism detection platforms should be able to have confidence in a system that analyses the author's use of language, rather than their choice of sources and aptitude for rewording existing work. Academics should also have more confidence that work hasn't been completed by another student or commercial service. Students should be able to have less of being penalised based on a single score, with a better understanding of the analysis that has taken place on their work. They should also be able to submit their work in an easy-to-understand manner, with as little work required on their part as possible. Platform providers should be able to reduce their costs by relying less on commercial databases for access to academic sources with which to cross reference.

The problem above should be considered adequately solved when the platform is able to identify instances of plagiarised text without referring to an external source – for example, to check against a database of known documents. Other acceptance criteria revolve around the use of the system: a user should be able to register and upload documents, and the system should be able to sufficiently analyse these documents, and provide additional insight to the user.

7 Literature Review

7.1 Existing Solutions

Product	Cross-reference papers from same cohort	Online Tutorials/Documentation	Timely report generation	Homoglyph detection	Multiple file upload	Cache webpages used as sources	Stylometric analysis	Multilingual	Database of source materials
Urkund [9]		✓			✓			✓	✓
TurnItIn [10] [11]	✓	✓	✓	✓				✓	✓
Copyscape Premium [12]	✓	✓							
PlagAware [11]	✓							✓	
PlagScan [11]								✓	✓
CheckForPlagiarism.net [11]	✓					✓	✓	✓	✓
Lexspec (Project implementation)		✓	✓	✓	✓		✓		✓

7.2 Analysis of existing commercial solutions

An analysis of existing solutions in this space indicates TurnItIn to be the most fully featured. TurnItIn's platform is also used by 98% of UK universities [10]. This makes TurnItIn the most popular plagiarism detection platform used by UK universities, and therefore the de-facto platform to compare against when developing a new solution.

The only competitor to TurnItIn in terms of features offered and widespread use is Urkund. Comparing the different solutions currently available relied heavily on third party sources such as comparisons and reviews of plagiarism detection tools by academics, as many existing solutions are proprietary, commercial systems that require a payment or subscription to use. This is especially true of TurnItIn, which typically only sells to large organisations such as universities.

CheckForPlagiarism.net is the only service that advertises analysis based on stylometry (sentence structure, use of grammar, etc). This area is intended to be a focus for the project's proposed system, and the fact that only one major commercial operation advertises this feature means there may be large scope for development and improvement in this area. On the other hand, it is possible that other platforms utilise this technology, but don't advertise its use.

Based on the analysis above, the ongoing success of the implementation of the project will be measured against the existing features and usability of Urkund, TurnItIn and CheckForPlagiarism.net.

The project implementation will aim to have feature-parity with TurnItIn, with the exception of multi-language support. The scope of the project only provisions for English language support, for both the system's interface, and its document analysis features. Attempting to match TurnItIn's extensive library of proprietary documents is likely a fool's errand, with very little in the way of documents from students to sample, and no agreements with commercial libraries and publishers (eg: JSTOR). On this basis, it's unlikely that the project could compete with an enterprise solution like TurnItIn based solely on library size.

Based on this, the project will instead use a combination of web crawling and public API access in order to index popular documents. As each document is submitted, a search across these public APIs will be made for relevant documents based on keywords lifted from the submitted document. Once these have been indexed, a traditional plagiarism analysis check will be carried out, attempting to identify passages in the submitted document that have been lifted from external sources. In order to make the project effective when faced with competition from other solutions with far wider reaching document libraries, and agreements in place with academic publishers, this project will focus particularly on using statistical and stylometric analysis of submitted documents in order to identify plagiarised or suspicious sections.

7.3 Analysis of research around plagiarism detection

Authorship verification and lexical analysis is an increasingly well researched area, with numerous papers already available covering many areas of discussion. Some papers of particular note for this project are detailed below.

Chow et al. have detailed a method of identifying anomalous passages based on classical cryptanalysis methods: they use statistical tests to determine whether a section of text deviates from the expected letter distributions for a given language, and if so by how much [8]. While this paper primarily focuses on text in the English language, the methodology is universal and could be applied to other languages with little effort. This makes it particularly suitable for a plagiarism detection platform that could, in the future, span several target languages.

Norvig has carried out work identifying the expected distributions of letters in the English language based on a large dataset from Google Books [13], covering a variety of different genres and writing styles. This builds on the methodology employed by Mayzner et al. in the 1960s, who manually counted letter use in various publications: magazines, periodic journals, news articles, to establish an expected table of letter distributions [14]. This information can be used in combination with the approach outlined by Chow et al. to detect anomalous passages.

Stylometry is also an important field when considering plagiarism detection. Stylometry covers the study of features in text in order to discern some meaning and pattern. Neal et al. cover some of the different features that are of significance when studying documents in their 2018 paper on the topic [15], which may prove useful during the implementation phase of the project. It is likely that features that have already been identified as meaningful by other academics will provide more valuable insights and better performance overall by the platform, compared to attempting to devise and extract new features from the text. To this end, a review of existing features will be informative.

8 Proposed solution to building a plagiarism detection platform

Many technical solutions are possible in order to attempt to solve the problem defined in the problem articulation section. This section will attempt to analyse and validate the choices made against other options.

The first possible approach would have been some form of desktop application that would be capable of analysing documents on a user's workstation. This approach was quickly ruled out for several reasons. Firstly, it would not be comparable with existing solutions in this space – which are all web-based and allow access from anywhere in the world. Additionally, it would limit the potential to use data to improve the performance of the platform, as well as require additional work on the user's end in order to install and update the application.

While this would appear to only leave a single approach: a web application, the underlying details of this implementation are a little more complex.

A traditional approach would result in a single service being deployed to a server, responsible for servicing all of a user's requests. In this particular case, this would involve a single service that presents one or more interfaces to the user, processes their uploaded documents, performs all the relevant analysis, stores the results, and then presents them to the user when requested. This is known as a "monolith" model.

While a monolith approach is less complex to implement, and results in a simpler system overall, there are several drawbacks – particularly when discussing web-based systems. Firstly there is the problem of tightly coupling different components together. This results in two loosely related systems (for example: the user authentication system and the language classification system) becoming intertwined, to the point where an internal change to one system impacts the other.

Tight coupling is often the choice of poor or rushed design decisions. It frequently results in a quicker initial implementation, and later down the line refactoring, bug fixes, and new feature implementations are difficult due to the ill-defined interactions between different sections of the codebase.

In addition to ongoing development and maintenance issues as a result of tight coupling, monolith applications are difficult to scale to meet user demand. As the application is completely self-contained, it's impossible to only allocate extra resources to problem areas (e.g. specific calculations). As a result, applications are often scaled vertically, which quickly becomes cost inefficient.

Based on the factors outlined above, it was decided that the platform implementation would follow a microservices model: instead of one self-contained application, the platform is composed of several independent services which interact with one-another. Each service is a fully independent codebase, and interaction is done by means of well-defined protocols – frequently an exposed API or work queue.

Despite the initial increased implementation workload for a microservices model, it was chosen due to the ease with which the platform can be later maintained and extended.

While more work will be required to setup a working framework: accept and store documents, store and present test results, so on and so forth, the modular nature of the design means that changing or adding new test services to the platform at a later date will be simple, and not require changes to any other component of the platform.

In addition to extensibility and maintenance benefits, a microservices model also allows for polyglot development: that is, picking a language based on its suitability for the task at hand, rather than trying to pick a language on a “one size fits all” basis. To this end, several languages will be used in the project’s implementation: Python will be used for data processing and analysis needs, C# is used where a simple API is required, while TypeScript is used to meet UI needs.

Finally, a distributed system allows for improved resilience and scaling capabilities. If a single component of the system fails (for example: a specific test service), the rest of the system remains available and usable. Similarly, individual services can be scaled when required – the PDF parsing service could be scaled up if a sudden influx of PDF documents is encountered and scaled down again when demand has subsided. Meanwhile, other components remain unscaled, allowing for cost savings as far as operating costs are concerned.

In order to support the deployment of the numerous services, as well as communications between them, a container and orchestration platform was chosen. For this project, each service was packaged into a Docker container, and deployed to a Kubernetes cluster. The Docker container ensured that all of the application and its dependencies were runnable as a single unit anywhere, and the Kubernetes cluster controls where each service is deployed and manages the communication between them.

The solution approach will be validated when a platform consisting of several services is deployed, with each service performing its own tasks. The platform should then present an ergonomic user interface to the end-user, which ties all of these services together. Tests should be carried out to ensure that a single service failing does not impact the other components of the platform.

9 Implementation

9.1 Data Storage

Across the Lexspec platform, several different types of data are stored, some of which are generated by several different services.

Given the distributed nature of the platform, there was no added convenience to only storing data in a single database (typically a RDBMS database). Based on this factor, several datastores were implemented in the platform.

Due to the distributed nature of the platform, and as mentioned previously, in some datastores data is generated by several different services, it was deemed unnecessarily complex to have several different services connecting to the same datastore directly.

To mitigate the issue of having several services tightly coupled to the underlying structure of a datastore, an encapsulation principle was applied, similar to the encapsulation principle present in OOP. The datastore itself became a separate distinct service, with a public API exposed to store and retrieve data. By encapsulating the datastore within its own API, all communications to the datastore followed the same well defined specification, with the potential for introducing future changes in a backwards-compatible way. This also simplified services consuming data from the store, as they no longer needed to include configuration and libraries to directly connect to the datastore. Instead, they could use HTTP requests to fetch and store data.

9.1.1 Document Storage

Document storage has been split into two separate classes. One service and datastore stores the document's metadata, while another exclusively stores its content.

9.1.1.1 Document Content

Document content is stored in an Elasticsearch cluster. Elasticsearch was chosen over a traditional RDBMS due to its aptitude for storing free text information, as well as well-documented replication and high availability options.

In addition, Elasticsearch has excellent statistical analysis capabilities. This meant that if any of these features were required by the platform (for example: relevance scoring), the existing functionality in the Elasticsearch cluster could be harnessed. What's more, there is no need to use a relational database to exclusively store content, as there's nothing typically relational about the information.

Initially, document parsers submitted text directly to the Elasticsearch cluster. It was decided this wasn't the correct approach, particularly as it meant any changes to the Elasticsearch cluster (or indeed, a complete change in underlying datastore), would require changes to several parsing services.

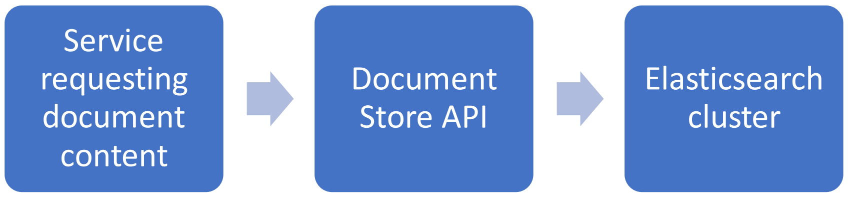


Figure 2 Final workflow for retrieving document content

The final implementation includes a C# API that sits in-front of the Elasticsearch cluster. Documents are submitted using HTTP POST requests, and retrieved using HTTP GET requests. By placing an API in front of the Elasticsearch cluster, any changes to the underlying cluster don't require code changes across the platform, and the API could, in the future, enable caching and other more advanced capabilities when storing and retrieving documents.

9.1.1.2 Document Metadata

A PostgreSQL database is used to store document metadata, this includes:

- Document name
- Random file name generated for the document once stored
- Which user uploaded the document
- When the document was uploaded
- The SHA1 hash of the document

This information is exposed by a simple Django Rest Framework project, and this API is consumed by several different services across the platform. Information is retrieved and stored by the API using HTTP requests.

9.1.1.3 Test Types, Test Submissions, Test Results

Different types of tests, submissions of each test (relating to a particular document), and the result of these test submissions are all stored in a PostgreSQL database. All of the data is accessed through a Django Rest Framework API.

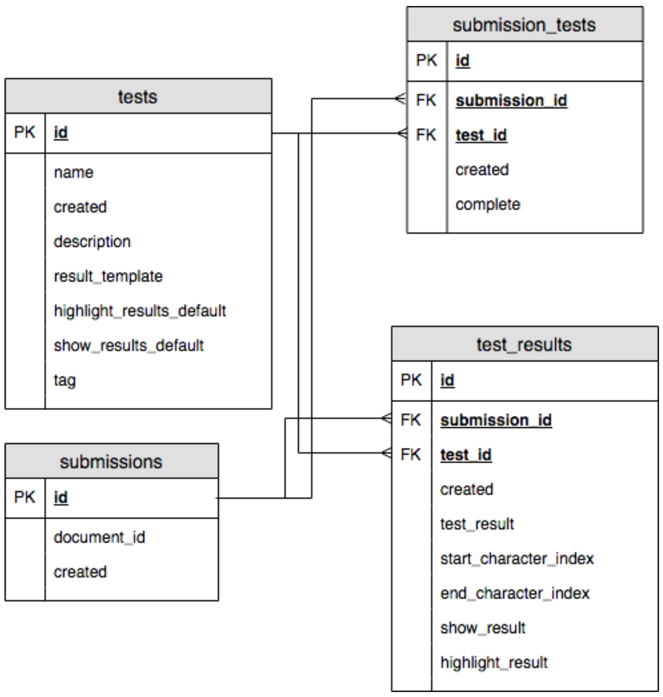


Figure 3

By storing this data in a normalised form in a relational database (see figure above), it is easy to query the data in a way that supports the platform's primary activities. An example of the type of query required for effective use of the platform is being able to list all of the test results for a particular submission. Had this data been stored in a NoSQL database (for example: MongoDB, or Elasticsearch), this kind of query would be more difficult to craft, and computationally more expensive to execute.

9.1.1.4 User Data

User information is stored in a PostgreSQL database. This database is only directly accessed by the identity service in order to authenticate and authorise a user. Other services use the OpenID Connect flows to retrieve information about a user over HTTPS.

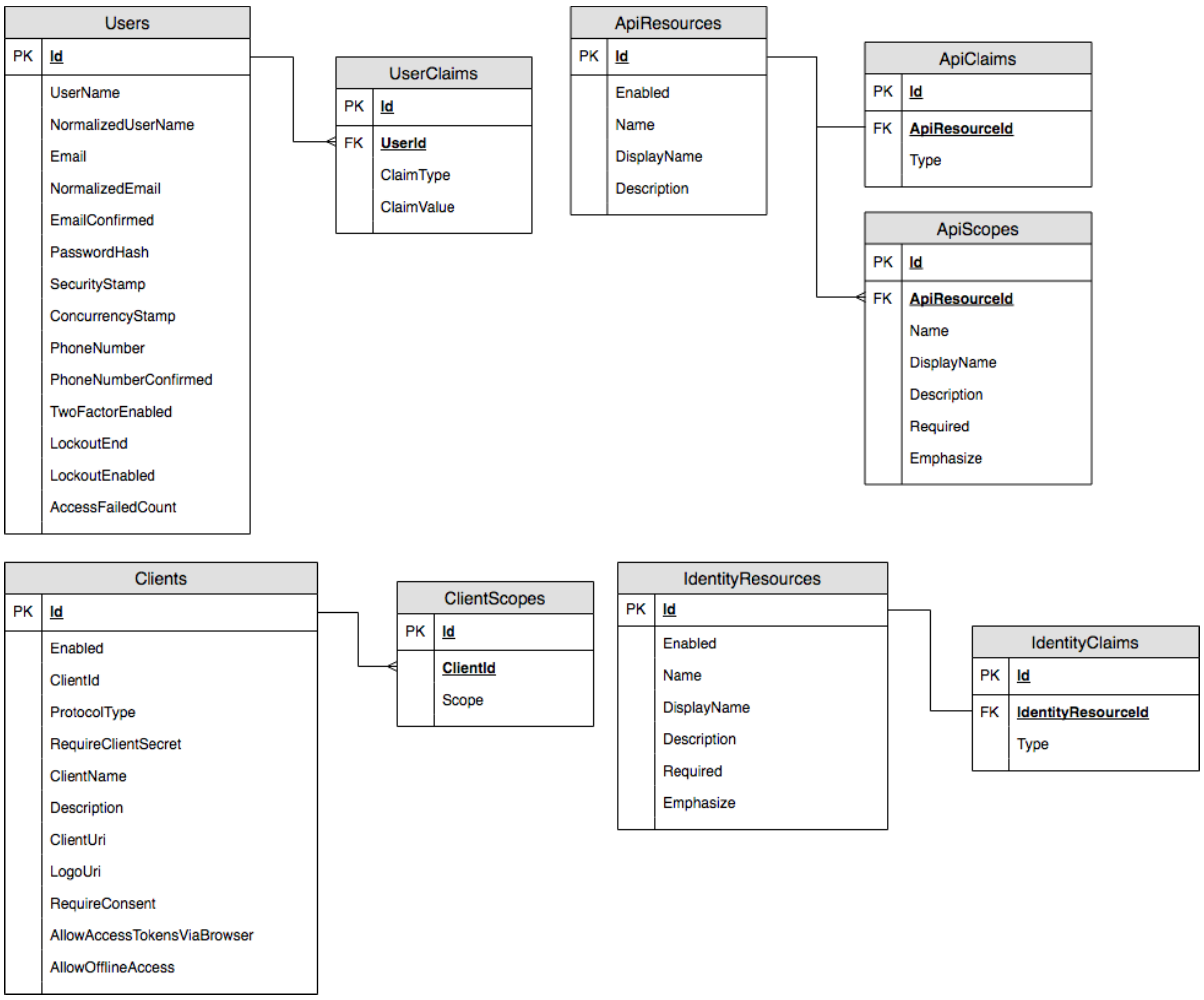


Figure 4 OpenID Connect Entity Relationships

User information stored by the identity service includes:

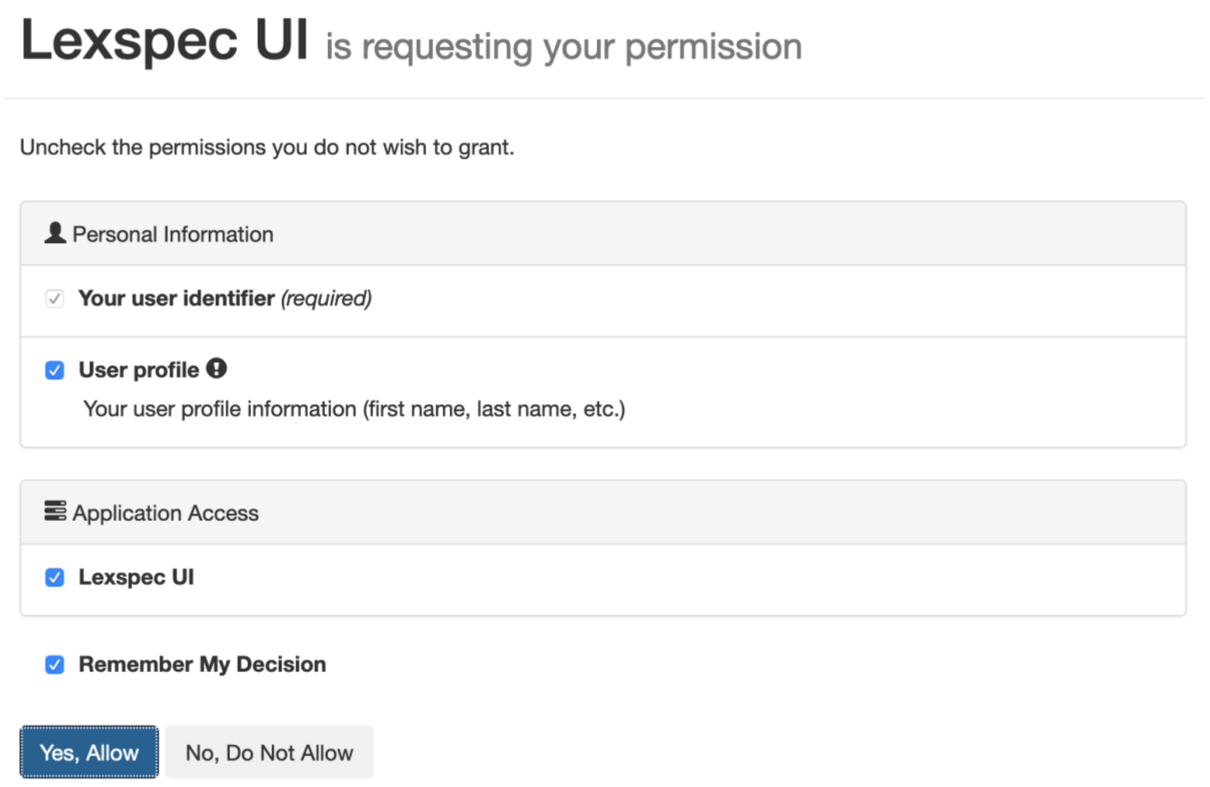
- Name
- Email address
- Hashed password
- User claims
  - Whether the user is an administrator or not
  - Personal information (birthday, address, etc)

While a variety of personal information is stored about the user, not all of this information is accessible to every service. Each service is only designated access to specific claims (for example: the user's email address, but not their birthday). This is specified as part of the OpenID standard. This can be seen in the entity relationship diagram above, where the information is segregated to allow fine-grained access:

- Authentication information is stored in the "Users" table
- Personal information is stored in the "UserClaims" table

- Clients are how services and APIs interact with the identity service. Some of the information relating to a client is detailed in the diagram above. This information is stored in the “Clients” table.
- The “ClientScopes” table represents what information a client can access
- Clients access resources, they do not access claims directly. The “IdentityResources” table lists resources relating to a user’s personal information (eg: their profile). A resource can be made up of one or more claims.
- The “IdentityClaims” field lists individual identity claim types, and which resource they are associated with. For example: a user’s address, birthday and telephone number may all be associated with the “profile” identity resource
- Just like identity resources represent personal information that a client has access to, API resources represent services that a client has access to. These are stored in the “ApiResources” table
- API scopes can be used to further control access to a service. For example, an API may have a “read-only” scope and a “read-write” scope. This may allow some clients read-write access to an API, but others read-only access.
- API claims dictate which claims are passed to the API as part of the access token. This may include information like the user’s email address, and their permission level.

The OpenID Connect standard also enables users to explicitly consent to which information is provided to some applications. Applications registered with the Identity Provider can choose whether or not consent is required to access the claims they are requesting (see the “Clients” table in the diagram above).


Lexspec UI is requesting your permission

Uncheck the permissions you do not wish to grant.

Personal Information

✓ **Your user identifier** *(required)*

✓ **User profile**
Your user profile information (first name, last name, etc.)

Application Access

✓ **Lexspec UI**

✓ **Remember My Decision**

Figure 5 OIDC Claims Consent Screen

In cases where consent is not required (typically core applications developed by the company – in this case, the Lexspec UI and administration UI), the application is given access to these claims as soon as the user is authenticated. In cases where consent is required (for example: a third-party integration trying to access some data from a user's profile), the user is explicitly asked whether or not they consent to sharing this information. Figure 5 captures this.

This approach to storing and accessing user data aims to empower the user to protect their own privacy, as well as be aware of what information is stored and shared by the platform. Core services of the Lexspec platform (the UI, administration UI, so on) operate on a "principle of least privilege" [16]. If a service does not require a user's email address, it is not shared with that service. This limits the surface area from which user data can be exposed, and enables the platform to practice good security hygiene.

9.2 Authentication and Authorisation

In order to implement an effective authentication and authorisation strategy for the platform, it was decided to use an implementation of the OpenID Connect (OIDC) standard. This standard is built on the OAuth 2.0 standard and provides for both authentication and authorisation flows.

Authentication and authorisation aren't considered a primary activity of the platform – rather, it is a supporting function. Based on this, and the complex nature of the standard, existing implementations of the specification were investigated. The requirements for an existing implementation were as follows:

1. The implementation must be open-source
2. The implementation must be under active development
3. The implementation must be well documented
4. The implementation should have an active community

Given the distributed nature of the platform, the implementation language of the OpenID Connect framework wasn't a major factor. Based on the requirements above, the following implementations were evaluated.

9.2.1 IdentityServer4

An implementation of the OpenID Connect standard developed in C#, with native support for the modern .Net Core platform. IdentityServer4 is certified by the OpenID Connect Foundation [17], and also has an active community online. The project is under active development, with sponsors including Auth0, and has detailed documentation on both IdentityServer itself, as well as areas of the OIDC standard.

IdentityServer4 has a barebones template project, from which a more fully featured platform can be developed. The barebones project includes a basic user interface for login and registration flows. Across IdentityServer3 and IdentityServer4, there's an active community online, with many blog posts and other resources available online to aid with implementation.

9.2.2 pyoidc

An implementation of the OIDC standard in Python, this implementation is open-source, but the project's README immediately outlines several problems: the documentation is out-of-date and lacking, there are few examples to use as reference, and the project is lacking active maintainers.

Based on the factors above, pyoidc was not chosen.

9.2.3 ORY Hydra

Initially, ORY Hydra appears the most feature-rich of the solutions evaluated here, with an extensive ORY ecosystem of other security tools on offer. However, ORY Hydra does not support a user management system (a system that stores user credentials, manages registration, so on and so forth), and additionally the documentation was found to be less detailed than that of IdentityServer4.

9.2.4 Final Implementation

Based on the factors above, IdentityServer4 was chosen as the framework on which to implement authentication and authorisation on the Lexspec platform. The two primary reasons for this choice were the detailed documentation available, and the barebones starter project that included a basic UI to handle login and registration flows. Additionally, the author's familiarity with the C# language and ecosystem was a point in favour of IdentityServer4.

The barebones IdentityServer4 project was implemented as proof-of-concept. Once a basic login and registration flow was successfully tested, the project was extended. Support for a persistent datastore (in this case: PostgreSQL) was added by following guides in the IdentityServer4 documentation [18].

Next, the administration interface was extended to enable creation of clients, API resources and identity resources. Previously this had to be done explicitly through code. It was decided to take the time to implement this feature in the user interface because at this stage of the implementation it was not known whether or not all microservices would need to support authentication and authorisation. If all services required authentication and authorisation, each service would need its own client and API resource designated in IdentityServer4.

9.3 Document Ingress

It was decided early on in the implementation process that there would be a single path available to ingress new documents into the platform. This single path would be used regardless of the document's format, or its origin. This decision was taken to ensure that this critical feature of the platform was implemented in a manageable and well-defined way, rather than having multiple ill-defined and poorly documented methods of storing new documents used in varying places across the platform.

Google's Golang was chosen as the implementation language for this service. Golang was chosen for its relatively simple syntax and its favourable performance in benchmarks compared to Python [19]. The ease of implementation of this relatively simple service would act as a bellwether for the wider use of Golang across the platform.

While the implementation of the ingress service was successful, during its development it seemed that this was in spite of the use of Golang, rather than because of it. As a lower-level language than Python, more work was required to extract form data and handle HTTP requests. Upon reflection, choosing Golang due to its better benchmarking performance was a case of premature optimisation, compounded by the author's own unfamiliarity with the language.

All documents are ingressed into the platform by submitting a HTTP POST request with the document data attached in the "document" field, regardless of that document's origin or file type. The ingress service then calls any other services required to correctly store the document in the platform's databases. Specifically, the ingress service calls the document metadata API via a HTTP POST request to ensure an entry is made into the document metadata store. This provides the document with a unique document ID, and also allows other services to discover the document and list it throughout the platform.

Finally, a RabbitMQ exchange is used as a broker between the ingress service and the file format parsing services. The document's file type is determined by its extension (for example: .pdf), and the document's metadata is then sent to a queue listened on by the relevant parsing service.

9.4 Content Parsing

Content parsing is achieved using individual microservices for each file-format. At the time of writing, services to parse plaintext and PDF documents have been implemented. In order to cope with fluctuating parsing demands, requests to parse a document's text are implemented using message queues. By using message queues, multiple instances of a parsing service are able to run at once at peak times (for example: when a dissertation deadline is due) – it also allows requests to be queued and the parsing services themselves to handle each request at a sustainable rate. Content parsing is particularly suited to background processing using message queues, as there is no immediate need for a document's text to be parsed once uploaded. Instead, this can be done in good time – alleviating load on the cluster. Additionally, in cases where a parsing service is unavailable or crashes, the requests to parse a document's content are not lost. They can instead be requeued, or re-delivered to another instance of the parsing service.

By communicating via message queues, rather than directly via HTTP requests, the coupling between document parsers and the initial file upload is loosened. New file formats can be supported simply by adding a new parsing service that listens on the appropriate queue. Scaling demands can be met by increasing the number of instances of a given parsing service if demand is increased – this is transparent to the ingress service itself, and requires no behaviour change or load balancing in either the parsing service or the ingress service.

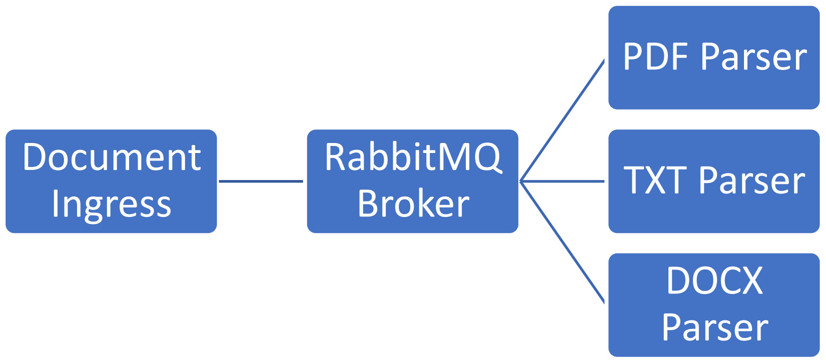


Figure 6

Figure 6 shows how the individual parsing services are completely abstracted away from the ingress service using the RabbitMQ broker. The document ingress sends a message to the RabbitMQ broker in a “fire-and-forget” manner.

It’s this abstraction that allows for easily implementing new file types, as well as scaling existing services for any reason.

By splitting document parsing services into their own individual applications one can benefit from polyglot development and choose the best tool for the task at-hand. As an example, a particularly capable PDF parsing library is available for Python, however it may be that the tools required to parse content from Microsoft Word’s docx format is more readily available in C#. Rather than compromising on an inferior toolset, or setting up an intricate system where an application in one language calls an application written in another, well-defined domain requirements (“parse a PDF document”, “Parse a docx document”) can be implemented using equally well defined services. A supplemental benefit of this approach is that any problem parsing a particular document format will not affect the platform’s support for another format, improving reliability.

9.5 Wikipedia Content Ingestion

In an effort to source additional material for model training, a simple Python service was created which ingests random Wikipedia articles at regular intervals using the document ingress service.

Unfortunately, the service appears to be rate-limited or otherwise blocked by Wikipedia, as requests started failing after a day or so of requests at 1 page request per 30 seconds. Before this point, several adjustments were made to the service: limiting documents to English language documents, and ensuring that Wikipedia pages to be ingested were longer than a certain length.

While it’s likely possible to debug the service and establish the network connectivity issue to the Wikipedia servers, enough data was ingested for development purposes, and it was decided that the time would be better spent implementing additional test services and correcting UI bugs.

Content ingestion from this service also served as a proof-of-concept for ingesting from other sources. One feature suggested during the project initiation was crawling the web for keywords relating to a document and ingesting these sources in order to provide more relevant training data. While this service has shown this to be theoretically possible, implementing it in an effective manner would be more time-consuming than originally thought. Consequently, no further ingestion services were developed due to time constraints.

9.6 Text Mapping

Results from test services are posted to a central database – this database stores the start index for a suspicious passage of text, as well as its end index. These indices are then used to highlight suspicious passages in the UI.

An unexpectedly difficult problem to solve that wasn't considered during the design process of the platform was that of mapping character indices. When a document is ingressed, its text is stored as-parsed – no processing is done to clean or otherwise modify the text.

When displaying the text in the UI, some formatting is done to make it presentable to a user. An example of such formatting is replacing "\n" (new-line) characters with the HTML "<br>" (line break) tag. This is because HTML does not process new-line characters. Similarly, some test services modify whitespace and other characters for the purpose of cleaning data before it's processed.

Consequently, the character indices stored by the test store database are the position of these characters in the original raw text, which differs from their position in the HTML-formatted text. Similarly, the character indices found in some test services must be "translated" back into the original character indices of the raw text.

This translation between raw text indices, and modified text indices was originally done in an ad-hoc fashion in each service that required mapping between two sets of indices for the same body of text. This approach quickly proved problematic, with several edge cases being identified in each implementation, as well as it being increasingly difficult to ensure that each implementation would map the same collection of characters in the same manner.

In order to tackle this problem more efficiently and effectively, all text mapping code was extracted out of the services in which it was implemented, into a single C# application that could be called via an API. This approach to polyglot development, where shared code is extracted into a service, rather than a library for a language, is outlined by Monzo [20], and used effectively in their codebase.

By extracting all of the code mapping character indices between two variations of the same text into a single service, it could be ensured that all services were mapping text in the same way. While this meant that the mapping itself was consistent, it was still incorrect for a variety of edge cases.

By creating a single service, a more comprehensive test suite was able to be implemented, with tests written to cover different edge cases as they were discovered. The process of discovering new edge cases was decidedly arduous, chiefly revolving around loading different texts into the UI, running tests, and discovering where the mapping service failed to work.

Once a document with a failed text mapping was identified, investigation began to identify where in the text the mapping was failing. This involved looking at the generated mapping and comparing with the two texts until a discrepancy in the mapping was found. Once a discrepancy was identified, a simple test case was implemented that replicated the scenario in its simplest form. The text mapping code was then modified until this new test case passed, and additionally all the existing test cases also passed.

This approach to ensuring the text mapper worked correctly has its roots in a TDD approach to building the service. In several cases, fixing a new test case broke existing cases. It should be acknowledged that if this approach hadn't been taken, and instead an ad-hoc approach was taken to fixing bugs as they were uncovered, the service would have remained unreliable, as existing use-cases were continually broken by bugfixes and undiscovered due to the lack of unit testing.

Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**
Original	H	e	l	l	o	,	\n	w	o	r	l	d			
Modified	H	e	l	l	o	,	<	b	r	>	w	o	r	l	d

Table 1

Table 1 shows a simple example of the mapping required by the UI. The top row of the table denotes character indices, while the first row is the original text, and the second row the text formatted for HTML display.

In the simple example above, the characters from the word "world" have been flagged as suspicious. As such, the database entry for this test result references a start character of 7, and an end character of 11. The full range is highlighted in the above table.

Without a text mapping service, the HTML-formatted document would incorrectly highlight only the first two letters of the word "world". This is due to the shifting indices introduced by implementing line breaks. The table also highlights the characters that would be highlighted on the web page.

In order to tackle this problem, an algorithm was implemented that iterates through each character of the modified text, if it is different to the original character at the given position, it then iterates through the original text until the matching character is found. Finally, it checks that this isn't simply the next instance of the character in the text (covering cases where the character that is being searched for is not present in the original text).

The pseudocode for the algorithm described above is below:

```
OriginalCharacters = GetCharacters(OriginalText)
ModifiedCharacters = GetCharacters(ModifiedText)

Mapping = Map()
```

```
PointerInOriginalCharacters = 0

for i = 0; i < Count(ModifiedCharacters); i++ {
     MatchingCharacterFound = False

     CurrentModifiedCharacter = ModifiedCharacters[i]
     CurrentOriginalCharacter =
OriginalCharacters[PointerInOriginalCharacters]

     If CurrentModifiedCharacter != CurrentOriginalCharacter {
          FindMatchingCharacter()
          CheckNotNextInstanceOfCharacter()
          SetMatchingCharacterFound()
     }

     If MatchingCharacterFound {
          ModifiedIndex = FindIndexWithExpandedHtmlTags(i,
ModifiedCharacters)
          AddMapping(PointerInOriginalCharacters,
ModifiedIndex)

     }
}
```

Edge cases existed around the occurrence of duplicate characters consecutively, a character in the modified text that isn't present at all in the original text, a character in the modified text that's not present in the same position in the original text, but does occur elsewhere.

While initially it was desired to create a 1-to-1 definitive mapping of characters between the text variations, this proved unachievable when considering the number of edge cases, and the little information available about how the characters have been shifted, inserted or removed. Instead, the final implementation makes use of a 1-to-1 mapping as frequently as possible, reverting to heuristic methods in the case of certain edge cases – typically this involves checking the surrounding characters in both the original and modified text to establish the mapped position. This heuristic implementation is not flawless – initially it only compared the character immediately before and after the character being searched for – it was quickly discovered that this circumstance was more common than initially supposed, and multiple characters before and after are now compared.

9.7 Text Sampling

Initially, each test service contained code to split source text into a series of samples. This rudimentary code simply split the text every n number of characters, and returned a collection of the n character samples.

While the logic involved in this operation was not complex, it was duplicated across the different codebases in the platform. This meant that there was the potential for different services to sample text in slightly different manners, resulting in an inconsistent behaviour across the platform, and also increasing the likelihood of bugs being introduced.

Furthermore, the naïve approach taken by the existing text sampling implementations meant that the samples themselves were not reflective of the structure of the text. By sampling every n characters, samples often ran across paragraphs and sentences, reducing their utility to a marker. Finally, edge cases existed where the naïve sampling code would start or end a sample on a character that would be removed when the text was reformatted for display on a web page (frequently a new line character). This introduced a bug where not all test results would be displayed in the web UI.

In order to rectify the issues outlined above, a new service was developed: the text sampling API. This API would be called by all test services in order to generate samples from a given source text. The API would also have a full suite of unit tests to ensure that samples were generated in the intended manner, and new features were added to the text sampling API that hadn't been present in previous implementations: the text sampling API would only begin or end a sample on an alphanumeric character, and an option was added to "snap to sentence". In the case that this option is enabled, samples will begin and end where sentences are defined.

By moving text sampling logic into a distinct service, the codebases of the platform's test services are reduced in complexity and size, and duplication of code is eliminated. This allows the platform implementation to conform with the DRY software engineering principle as much as possible.

As a consequence of these changes, it is no longer possible to guarantee that a sample will be exactly n characters in length, but it was believed that small variations in sample size was a sensible trade-off for more informative samples that better reflect the underlying structure of the document, and samples that start and end on characters that are consistently included in all variations of a source text.

9.8 Language Analysis Services

As with content parsing, test services consume message queues in order to service requests – for many of the same reasons – namely: requests can be queued during peak periods to ensure the platform continues operating, in-demand test services can be scaled horizontally with no extra effort, and test requests aren't lost if a service crashes or otherwise malfunctions. Finally, it's not essential that all tests run immediately, this can instead be done over-time in the background.

The use of a message broker also decouples the delivery of test requests from the rest of the platform, meaning that each time a new test service is added, the only additional change is that the test's metadata is submitted to the administration panel. No other changes or updates are required to the platform's other APIs and services. This enables rapid iteration and development.

9.8.1 Homoglyph Detection

By implementing the method described by Chow et al. [8], homoglyph [1]detection has been added to the platform. Homoglyph detection relies on using index of coincidence values, as well as a statistical chi square test in order to attempt to detect anomalous excerpts of text. Homoglyphs are used by plagiarisers in an attempt to fool plagiarism detection platforms by changing characters in a word. While the word looks normal to a human, it may be made up of a mixture of Latin, Greek and Cyrillic characters, encoded differently and therefore composing a different word to a machine.

Homoglyph detection was implemented using a random forest classifier in order to improve detection rates. Supervised training was employed, with samples labelled as either containing substituted homoglyphs, or being unaltered, in order to better classify when a sample may contain homoglyph substitution.

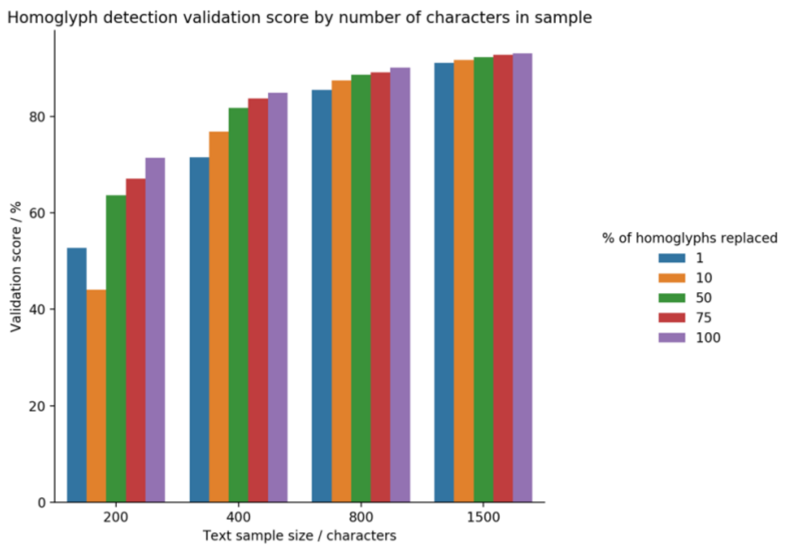


Figure 7 Model accuracy across different sample sizes and homoglyph replacement rates

[1] Homoglyphs are characters that are visually indistinguishable to a reader, but are in fact from different alphabets. For example: "a", " а " and "a" are all different characters – despite looking nearly identical.

Figure 7 shows the accuracy of this model across different text sample sizes, and with different percentages of homoglyphs substituted in the training data. Unsurprisingly, accuracy improves as both sample size and percentage of homoglyphs replaced increases. Some unexpected observations are how substituting 10% of homoglyphs in a 200 character sample leads to substantially worse training outcomes. The author is unclear as to why this is.

While longer samples of text produce more accurate outcomes (with 1500 character excerpts typically being correctly classified over 80% of the time), the method used to alter the original samples and insert homoglyphs may not be representative of how an author may substitute homoglyphs into their own work. The current methodology replaces X% of all homoglyphs in the source text (eg: 10% of every 'e' character, 10% of every 'a' character, so on). The training of this model may be improved further in the future by developing a more "natural" way to substitute these characters. Additionally, while longer samples of text provide more accurate results, they are less precise than shorter samples, meaning that the information conveyed in the document's analysis report is less useful to an end-user.

This same technique also detects white characters placed between words – a different means to the same ends of presenting one word to a reader, and another to a computer. An example may be "this is a sentence" becoming: "thisiisiaisentence" – where each "i" between words is coloured white: "this is a sentence". While not looking suspicious to a human reader, the sentence is nonsensical when every character is read.

9.8.2 Punctuation Analysis

The punctuation analysis service analyses the frequency of different punctuation marks across a sample of a given size. The size of the sample is also used as a feature in the final model.

Punctuation frequency was chosen as a feature to model after the author noted that even associates in the same cohort often use punctuation in a unique way, to the point where in a report authored by several people, it's frequently possible to distinguish authors by eye after analysing the frequency of common punctuation marks such as the comma. This observation is backed up by several academic papers [15] [21], which highlight punctuation use as a syntactic feature which can be modelled.

The following features were modelled:

- Comma
- Apostrophe
- Question mark
- Full-stop
- Sample length

After further review of the literature, the following punctuation marks could also be modelled, and were suggested by Neal et al. as indicative features of authorship [15]:

- Colon
- Semi-colon
- Double quote marks
- Exclamation marks

It is unclear without further investigation whether adding more punctuation marks would improve the signal of the model. It may be that measuring less common punctuation marks (for example: the semi-colon) introduce noise to the data set, with no clear indication of authorship.

The implementation of the punctuation analysis is done in such a way that adding and removing new features to the model is trivial, however there is no apparatus setup to measure the impact of changing features on the model's accuracy. In order to do this effectively, the changes would have to be tested against a wide variety of documents, and there simply aren't enough documents in the platform at this stage to support such widescale testing.

Several possible changes to the punctuation analysis model could be made, further investigation would be required to determine whether these changes would improve the model. The model currently uses an ensemble random forest classifier, due to the improved accuracy of this classifier when determining language classification. Following the success of using a one-class SVM when modelling lexical features, it may be appropriate to use the same classifier in the course of punctuation analysis, particularly given the relatively sparse nature of the dataset. In any case, hyperparameter tuning on both a random forest classifier and one-class SVM may produce more accurate results. Additionally, weighting occurrences of different punctuation marks may also provide a useful insight – perhaps the number of occurrences of a colon mark is more indicative of authorship style than the number of quotation marks used?

The punctuation analysis service suffers from frequent false-positives. This is particularly sensitive to the training data used. However, it did correctly flag plagiarised sections in a test document composing of both plagiarised and original passages. The sensitivity to training data further supports the argument that a one-class SVM may produce more accurate, and importantly, consistent results for this test service, because a one-class SVM is better suited to anomaly detection than an ensemble classifier, and in the case described above, anomaly detection is the ultimate goal.

9.8.3 Lexical Feature Analysis

Initially, the lexical feature analysis service only measured average word length, however some proof of concept testing showed that this feature was not distinct enough to produce useful results, with large swathes of a document known to be authentic being flagged as plagiarised.

After consulting the literature [15], the following features were instead chosen:

- Number of characters (all, uppercase, lowercase)
- Number of digits
- Number of whitespaces
- Number of words
- Average word length
- Average sentence length in words

These features were chosen as they were highlighted by a survey of stylometry techniques and applications [15] as being effective features for stylometry analysis using SVMs. Building on the recommendations made in this survey, the existing random forest classifier was replaced with a one-class SVM.

One-class SVMs focus on anomaly detection, returning only two outputs: +1 for inliers, and -1 for outliers. One-class SVMs differ substantially from other classifiers implemented, in that there are no labels for training data, and only one class is submitted for training. While other services made use of ensemble classifiers, with samples from a user's previous submissions being sampled as positive training data, and samples from other users' documents being submitted as negative training data, the lexical feature analyser trains using only samples from a user's previous submissions.

Compared to the previous ensemble classifier, the one-class SVM provides more consistent results, with the same sections of a document being flagged as plagiarised across multiple test runs. In comparison, test services using ensemble classifiers return wildly varying results for the same document across multiple test runs, due to the variety in training data used. The one-class SVM implementation also returns fewer false positives than the ensemble classifier. It is thought that with sufficient hyperparameter tuning, this accuracy rate could be further improved.

This service was tested using two variations of the same document. The first variation was the original document, known to be genuinely authored and free of plagiarism. The second variation of the document retained the same text, but added several paragraphs from another student's version of the same assignment.

When the original document was tested, several false positives were returned as having anomalous lexical features. When the plagiarised variant was tested, the same number of false positives were flagged, with none of the plagiarised sections being detected. It's possible that some hyperparameter tuning may be able to adjust the model so that these instances are correctly detected, albeit at the risk of a higher number of false positives.

Given that the authors of the original and plagiarised text have the same level of education, were writing for the same assignment, and have a similar command of the English language, it’s likely that there weren’t sufficient distinctions between the lexical features used in the original and plagiarised texts for the model to accurately distinguish the two authors in such short passages (samples in this test are taken at approximately 400 character intervals).

9.8.4 Language Classification

The first analysis service implemented was a tool which classifies the language of the given document. For the purposes of this project, only two classes were defined: ‘english’ and ‘unknown’.

As the first analysis service implemented, the language classification service served as a blueprint for further services in this area. Language classification was chosen as a starting point due to the relatively simple nature of the analysis required. This meant more work could be put towards general design decisions which would influence later tools.

It was decided to implement this tool using Python, due to the large number of libraries surrounding machine learning available in the Python repositories. It was believed this would make the analysis work simpler. In addition, rather than implementing a web server to accept test submissions, the service would instead consume a RabbitMQ work queue. Both of these design decisions would go onto form a basic outline for the test services that followed.

Subscribing to a work queue rather than listening to requests over HTTP was chosen for several reasons. Firstly, a work queue increases the resilience of the platform. If the language classification service crashes, or is unavailable for another reason, test submissions are simply queued by the message broker, ready to be consumed once the service is available again. Had a HTTP server been implemented instead, some logic would be required at the submitting service to handle failures to communicate with the language classification service.

Additionally, subscribing to a work queue provides ample opportunity to scale horizontally at a later date, if required. This can be done by having many instances of the language classification service running, each listening to the same work queue. The message broker will then deliver test submissions to each instance in turn, in a “round-robin” fashion. If a single instance fails, any unprocessed work that was delivered to that instance is redelivered to other instances. Had a HTTP server been implemented, a separate load balancing application would have to have been put in front of the language analysis services, and configured to send to each instance independently. This increases the complexity of the system, and also makes its configuration more brittle, as each language classification instance needs to be manually registered with the load balancer.

Language classification was implemented based on the work done by Norvig , who analysed ngrams extracted from a Google Books dataset in order to calculate the expected frequency of individual letters, as well as combinations of letters (known as unigrams and n-grams, respectively) [13]. From this, classic cryptanalysis methods were used to calculate the expected index of coincidence [22] values. This approach was detailed by researchers attempting to use classic cryptanalysis methods to detect spoofing in documents [8].

The index of coincidence calculation was implemented in Python, and initial tests using existing assignments as test data showed that the IoC value for these documents was in an approximate range of the value calculated by Chow et al. [8], however there were issues with this implementation that prevented it from being used independently for language classification.

Although the values obtained using this method were similar to the figure obtained by Chow et al., the values obtained varied depending on the document submitted, and were never within rounding distance of the "perfect" figure. What's more, it was discovered that the IoC value differed substantially depending on the length of the work submitted. Attempting to analyse individual sentences, or even paragraphs, produced mixed results – often the IoC value obtained was not close the suggested value of 0.066 for English unigrams [8].

In order to establish some sort of range that could be used to classify text as English, text from French literature and current news articles was submitted to the prototype service. As expected, this produced a measurably different result (of around 0.072) compared to English submissions, but suffered from the same flaw: as the corpus of text submitted shrank or grew, the IoC value obtained varied. For particularly short phrases, it was clear there was some substantial overlap with IoC values obtained for short English sentences.

For the prototype service, an attempt was made to implement a simple classifier to determine whether the text provided was written in English. This classifier calculated the index of coincidence for unigrams, the index of coincidence for bigrams, and then determined whether both were in a range defined as expected English values. A pseudocode implementation of the logic is below:

```
If bigrams_are_english(bigrams) and unigrams_are_english(unigrams)
    then return 'english' else return 'unknown'
```

While this simple logic occasionally returned the expected result, it more frequently than not simply classified the text as 'unknown' – undoubtedly due to the very rigid nature of the range established for both unigram and bigram index of coincidences.

With an initial prototype language classification service implemented – at this point the service could accept requests from the work queue, attempt to classify the text's language, and then post the results to a database, work started to improve the classification accuracy.

A large corpus of English text was required for training data. Initially, text from academic papers and similar materials was searched for, however it was too difficult to find a large and diverse corpus of text that was accessible in a programmatic manner. Usage of the CORE [23] database was explored, but it was found that the search tool didn't work as expected, making it impossible to automate the download of documents that were already known to be written in English.

Instead, large excerpts of text from Project Gutenberg were sourced. This resulted in around 100MB of English text data to be used for training and testing purposes. In order to implement a binary classifier (where the defined classes are ‘English’ and ‘unknown’), data from another language was required. Initially, an excerpt of French texts was chosen, also from Project Gutenberg.

In order to quickly prototype a working classification model, the Scikit-Learn Python library [24] was used. This library provides implementations of common models, and makes them accessible to developers.

It was decided to start a new project to separately train the model. This project would prepare the data, train and test the model, and then save the trained model. It was decided to separate this from the language classification service’s codebase in order to keep the latter’s codebase small and clean, and also provide expansion opportunities in the future, if the method of retrieving data becomes more complex.

Initially a logistic regression classifier was implemented. The training data was prepared by splitting it into 2000-character chunks, and then the model was trained on these 2000-character chunks. Initial results using Scikit Learn’s inbuilt cross-validation tools were promising, scoring a 96% accuracy rate when testing against the partitioned test data – however tests of real-world data showed that the classifier almost exclusively classified text as English, regardless of the true language – tests were carried out with texts of English, French, Dutch and Finnish origin of varying lengths.

Analysis of the training data showed that there was a large disparity in the amount of training data from each language. While 100MB of English training data was available, there was only around 3MB of training data from alternative languages. As a result, the model was hugely overfitting for English texts. The in-built cross validation tools showed accurate results due to the probability of selecting an English sample to test, given how many more English samples were available.

In order to attempt to rectify the imbalance in training data, the model was refitted with training data weighted based on the number of samples available from that class. This meant that despite there being more English samples to train on, they would be weighted as less important compared to samples from other languages. This brought the accuracy when testing to 40%. Next, the training data was changed so that there was around 3MB of data from each class to train on, equally weighted. This provided scores of around 50% accuracy when testing.

Several other simple classifiers were used: a Naïve Bayes implementation was tested, as was a stochastic gradient descent classifier. Both of these classifiers rendered an accuracy of 50-55%, depending on the model’s parameters. A K neighbours classifier was implemented, which resulted in tests showing around 60% accuracy.

After consulting the Scikit Learn documentation, an ensemble classifier was implemented. Ensemble classifiers “combine the predictions of several base estimators built with a given learning algorithm in order to improve generalizability / robustness over a single estimator”

[25]. Based on this, a random forest classifier was used. The random forest classifier produced results of around 70% accuracy. These results were slightly improved by changing the training data: the model was now trained based on sentences, in addition to chunks of text of varying lengths.

The model's features are the index of coincidence of English unigrams, English bigrams, and additionally the number of characters in the sample. By accounting for the number of characters, variations in IoC values introduced by length can be accounted for.

Real world testing of the random forest classifier showed that it only appeared to falter on phrases containing 40 characters or less. Even training the model on chunks of text of this size and smaller failed to improve accuracy rates, likely due to the small amount of information that can be gleaned from a sample of this size.

The exported random forest classifier was then implemented in the language classification service. The language classification service deserialises a Python object from the saved data, and this object is used to classify samples sent to the test service.

9.9 Administration UI

During development, it was becoming increasingly difficult to test the platform end-to-end – a complex mixture of port-forwarding, proxying and directly entering data into message queues was being used. In order to simplify development, an administration back-end was devised. The implementation of this back-end would serve two purposes: firstly, it would make testing the platform simpler, as well as enable effective monitoring of its current state. Secondly, it would serve as a prototype for the main end-user interface, as the end-user interface would contain a subset of the functionality of the administration back-end.

In order to effectively act as a prototype for the end-user interface, the administration back-end had to be implemented in the same way. To this end, the administration back-end makes full use of OpenID Connect to authenticate, and uses VueJS [26] – the same frontend framework that would be employed by the end-user interface. By making use of the same technologies and frameworks, code could be shared between these two interfaces, reducing ongoing development and maintenance requirements.

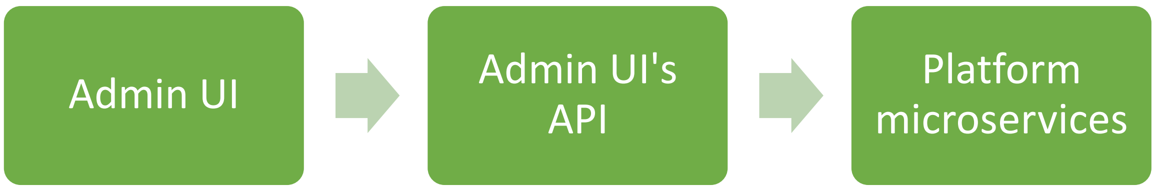
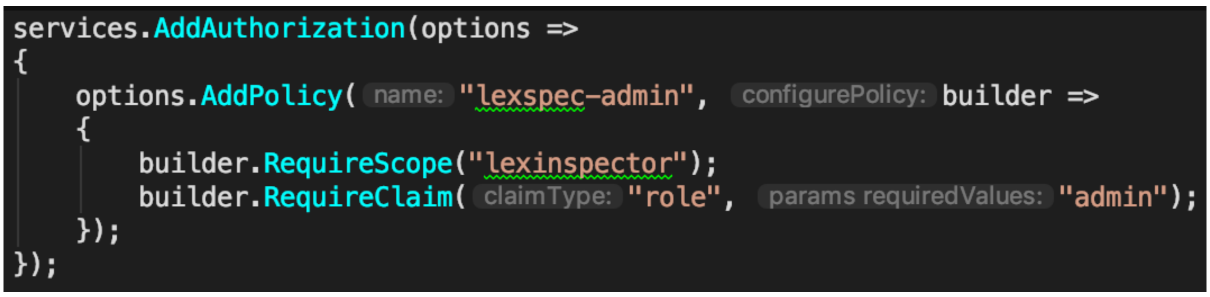


Figure 8 Workflow between admin UI and the Lexspec platform

The administration interface was implemented as an SPA (single page application). It relies completely on an API to interact with the Lexspec platform, with the overall workflow for interactions looking something like figure 8. Note that the admin UI does not interact with the Lexspec platform's microservices directly, but instead talks to an intermediate API. This is primarily done to allow the intermediate API to handle authentication and authorisation – the microservices that power the platform assume that all requests to them have already been correctly authorised in order to reduce complexity. In order to facilitate this design, the microservices themselves are unreachable from the internet thanks to a firewall, accessible only over a private network.

A key concern when implementing the administration backend was that it was only to be accessible to certain users, due to the sensitive nature of the content it provided access to: through the administration panel, platform configuration changes could be made, tests could be submitted, and any document stored by the platform could be viewed or deleted. Therefore, despite using the same authentication mechanism and account store as the end-user interface, it was important that not all users be able to access and use the administration panel.

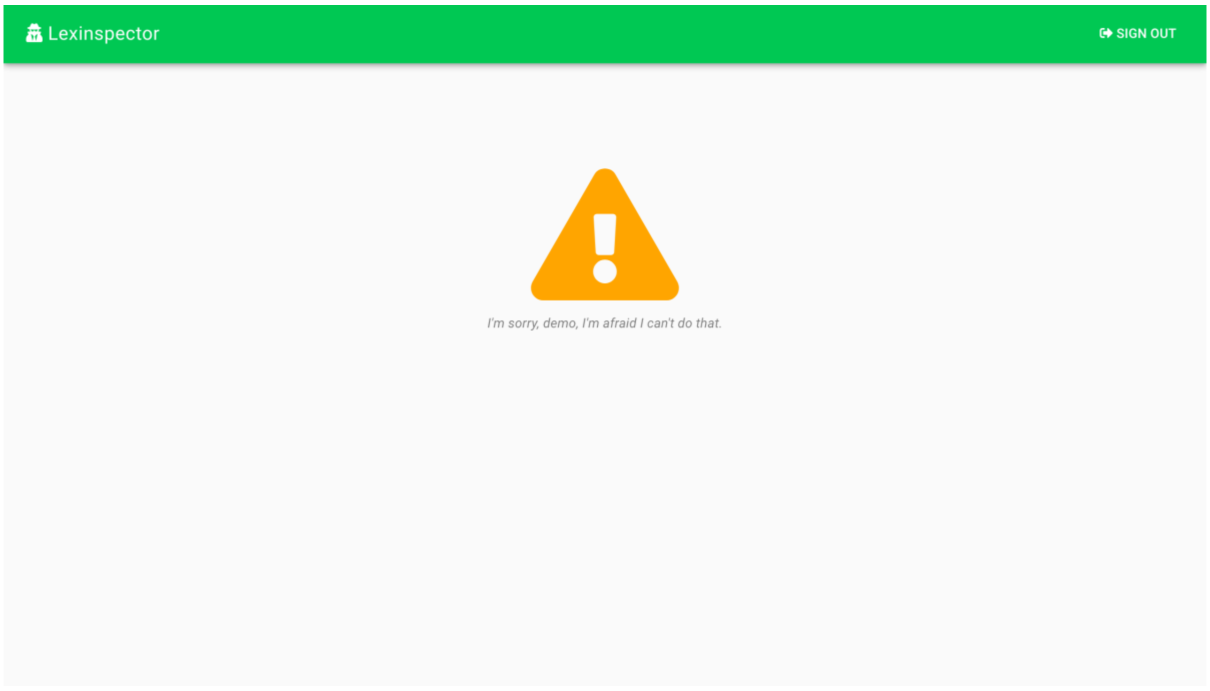


```
services.AddAuthorization(options =>
{
    options.AddPolicy( name: "lexspec-admin",  configurePolicy: builder =>
    {
        builder.RequireScope("lexinspector");
        builder.RequireClaim( claimType: "role",  params requiredValues: "admin");
    });
});
```

Figure 9 Middleware requiring an "admin" role in order to process requests to the administration API

In order to correctly implement this access control, middleware was added to the administration interface's API, whereby the role assigned to the access token provided to the API is checked. If the role is not "admin", the request is refused. This is implemented using the code in figure 8.

The administration panel SPA will redirect to the identity service for authentication if no access and identity token is stored in the browser – it will also redirect if these tokens have expired. As the SPA is fully implemented in client-side Javascript, it would be possible for a malicious user to interfere with the redirect process and instead explore the user interface, however without an access token no data will be returned from the API, making the interface useless.



I'm sorry, demo, I'm afraid I can't do that.

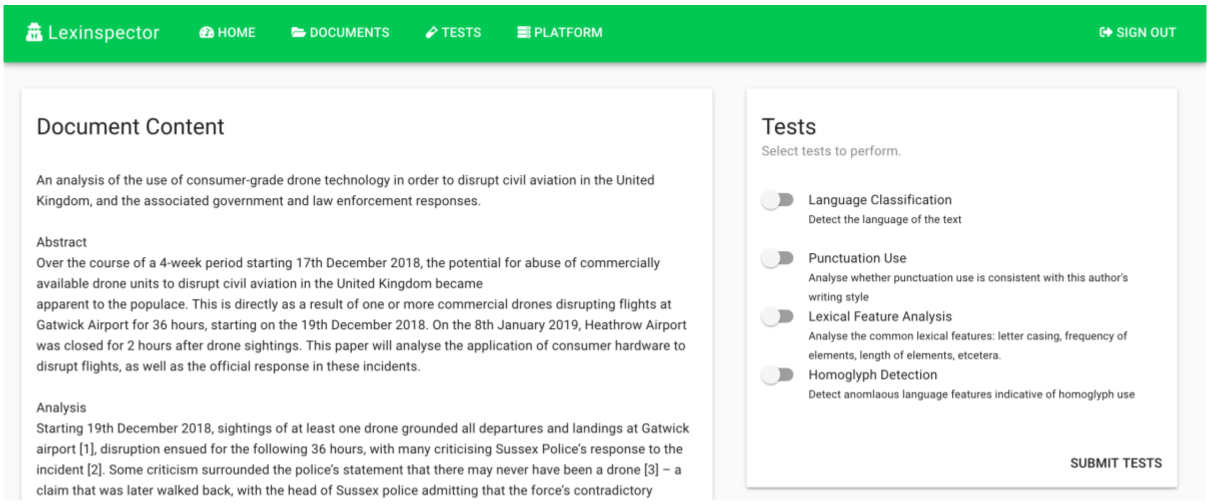
Figure 10 Unauthorized screen displayed to an end-user accessing the administration panel

If an end-user attempts to access the administration panel, they are presented with the screen in figure 10. In this case, an access token has been provided, but it does not contain the appropriate role or scope information to allow access to the administration interface.

Similarly, as each access token specifies a scope, it would not be possible to use an administrator's access token for the end-user interface to access the administration panel. By limiting the scope of each access token to only the resources required for the task at hand, the overall platform becomes more secure.

9.9.1 Document upload and test submission

Once an administrator has authenticated themselves correctly, they are able to use the administration panel for a variety of purposes. During development, it was used to upload documents and run tests. It can also be used to view the results of these tests, producing a similar report to the one displayed to users in the end-user interface.



Document Content

An analysis of the use of consumer-grade drone technology in order to disrupt civil aviation in the United Kingdom, and the associated government and law enforcement responses.

Abstract
Over the course of a 4-week period starting 17th December 2018, the potential for abuse of commercially available drone units to disrupt civil aviation in the United Kingdom became
apparent to the populace. This is directly as a result of one or more commercial drones disrupting flights at Gatwick Airport for 36 hours, starting on the 19th December 2018. On the 8th January 2019, Heathrow Airport was closed for 2 hours after drone sightings. This paper will analyse the application of consumer hardware to disrupt flights, as well as the official response in these incidents.

Analysis
Starting 19th December 2018, sightings of at least one drone grounded all departures and landings at Gatwick airport [1], disruption ensued for the following 36 hours, with many criticising Sussex Police's response to the incident [2]. Some criticism surrounded the police's statement that there may never have been a drone [3] – a claim that was later walked back, with the head of Sussex police admitting that the force's contradictory

Tests
Select tests to perform.

Language Classification
Detect the language of the text

Punctuation Use
Analyse whether punctuation use is consistent with this author's writing style

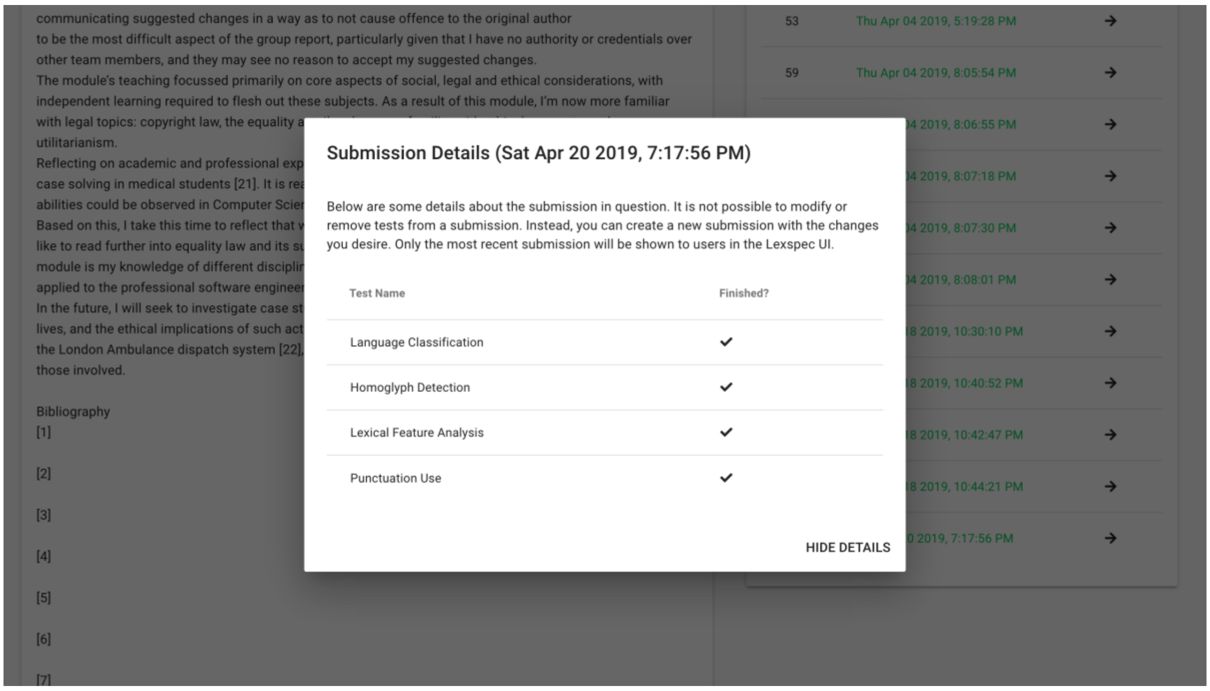
Lexical Feature Analysis
Analyse the common lexical features: letter casing, frequency of elements, length of elements, etcetera.

Homoglyph Detection
Detect anomlaous language features indicative of homoglyph use

SUBMIT TESTS

Figure 11 Interface used by an administrator to select which tests to submit for a document

An important distinction in this area between the end-user interface and the administration panel is that when a document is uploaded through the administration panel, no tests are run by default. Instead, an administrator can use displayed on the right-hand side of figure 11 to select which tests to run, and then submit these tests.



communicating suggested changes in a way as to not cause offence to the original author
to be the most difficult aspect of the group report, particularly given that I have no authority or credentials over other team members, and they may see no reason to accept my suggested changes.
The module's teaching focussed primarily on core aspects of social, legal and ethical considerations, with independent learning required to flesh out these subjects. As a result of this module, I'm now more familiar with legal topics: copyright law, the equality a
utilitarianism.
Reflecting on academic and professional exp
case solving in medical students [21]. It is rea
abilities could be observed in Computer Scie
Based on this, I take this time to reflect that w
like to read further into equality law and its su
module is my knowledge of different disciplir
applied to the professional software enginee
In the future, I will seek to investigate case st
lives, and the ethical implications of such act
the London Ambulance dispatch system [22],
those involved.

Bibliography
[1]
[2]
[3]
[4]
[5]
[6]
[7]

53 Thu Apr 04 2019, 5:19:28 PM
59 Thu Apr 04 2019, 8:05:54 PM
4 2019, 8:06:55 PM
4 2019, 8:07:18 PM
4 2019, 8:07:30 PM
4 2019, 8:08:01 PM
8 2019, 10:30:10 PM
8 2019, 10:40:52 PM
8 2019, 10:42:47 PM
8 2019, 10:44:21 PM
0 2019, 7:17:56 PM

Submission Details (Sat Apr 20 2019, 7:17:56 PM)

Below are some details about the submission in question. It is not possible to modify or remove tests from a submission. Instead, you can create a new submission with the changes you desire. Only the most recent submission will be shown to users in the Lexspec UI.

Test Name	Finished?
Language Classification	✓
Homoglyph Detection	✓
Lexical Feature Analysis	✓
Punctuation Use	✓

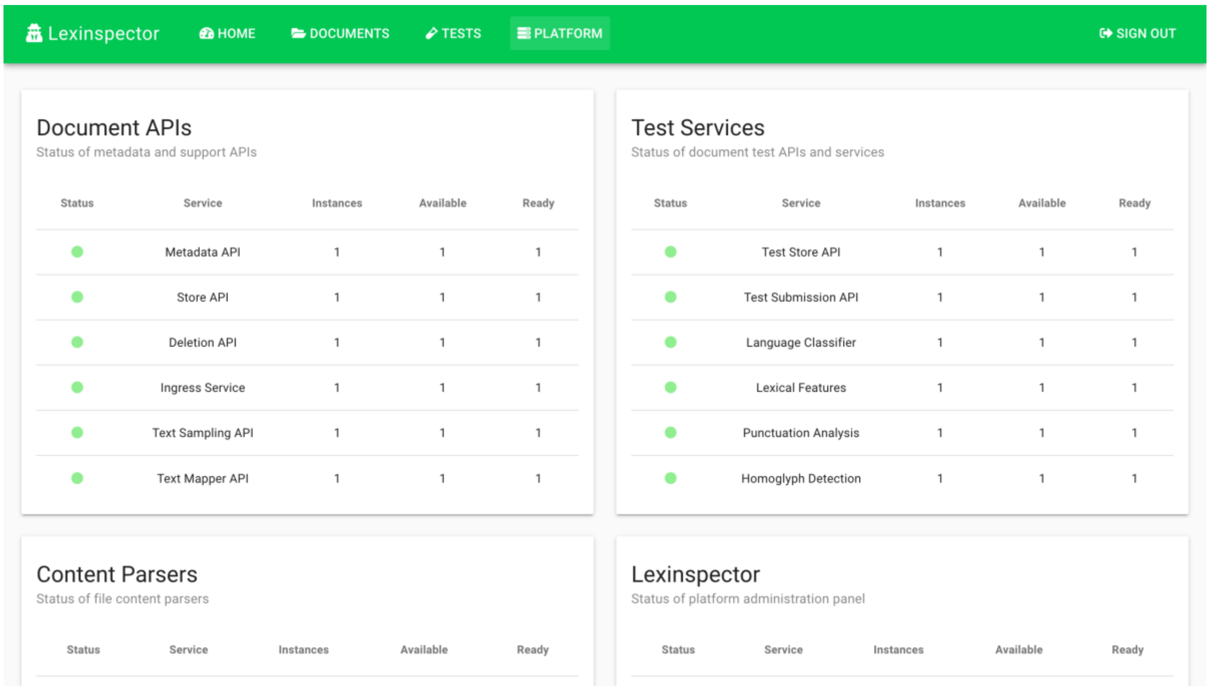
HIDE DETAILS

Figure 12 Tests submission status

Figure 12 shows the next step of this process. On the right-hand side of the figure, in the background, one can see the list of previously made submissions. Clicking on any of these submissions will display the foreground modal window, informing the administrator which tests have completed. The administrator can then view the results of the tests by clicking on the arrow to the right of the submission date and time.

9.9.2 Platform status

As the number of microservices powering the platform grew in number, it became more important to be able to tell at a glance whether or not all of these services were working correctly. To meet this requirement, a platform status page was implemented.



Document APIs

Status of metadata and support APIs

Status	Service	Instances	Available	Ready
●	Metadata API	1	1	1
●	Store API	1	1	1
●	Deletion API	1	1	1
●	Ingress Service	1	1	1
●	Text Sampling API	1	1	1
●	Text Mapper API	1	1	1

Test Services

Status of document test APIs and services

Status	Service	Instances	Available	Ready
●	Test Store API	1	1	1
●	Test Submission API	1	1	1
●	Language Classifier	1	1	1
●	Lexical Features	1	1	1
●	Punctuation Analysis	1	1	1
●	Homoglyph Detection	1	1	1

Content Parsers

Status of file content parsers

Status	Service	Instances	Available	Ready

Lexinspector

Status of platform administration panel

Status	Service	Instances	Available	Ready

Figure 13 Part of the platform status overview

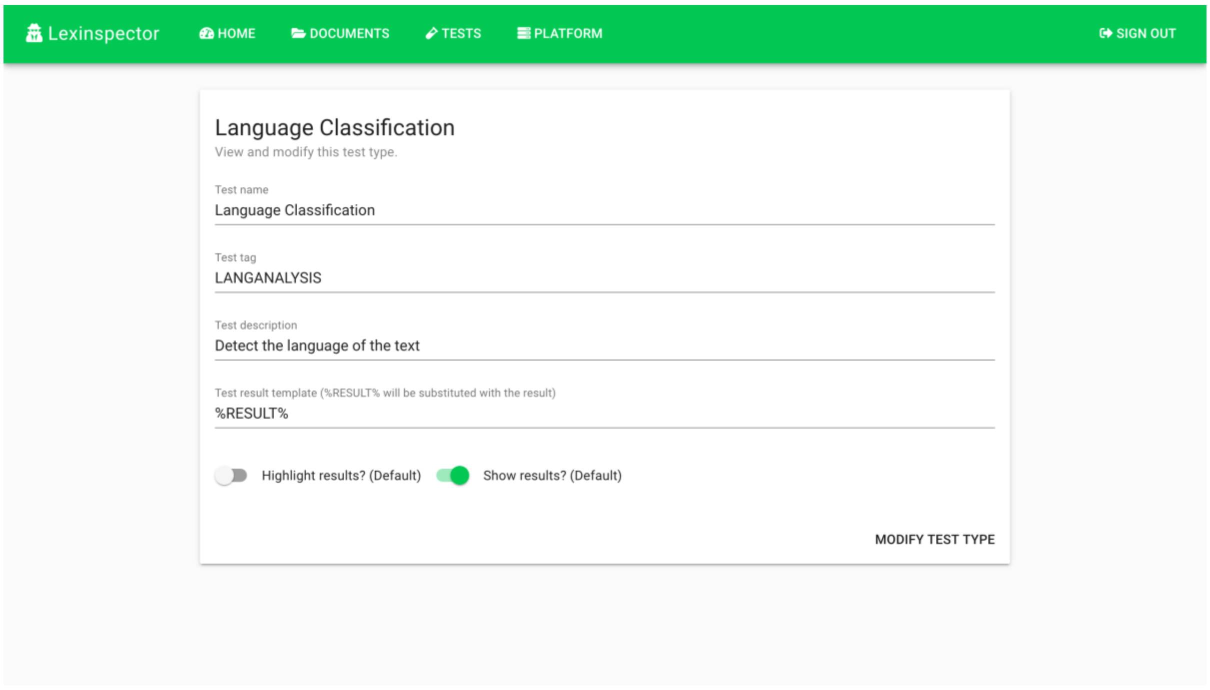
Shown in figure 13, the page tells an administrator at a glance how many instances of each service are running, as well as how many should be running. A simple traffic light system is used to convey information quickly:

- Green: Number of instances desired matches the number of instances ready
- Amber: Number of instances ready does not match the number desired, but is > 0
- Red: No instances of this service running

The status of each service is pulled live from Kubernetes [27], the container orchestrator that controls which services are running, how many instances of the service are running, and where they are physically hosted. The information is retrieved by the administration panel's API, authenticating securely using a short-lived certificate it receives from Hashicorp Vault [28] – a secrets management system. The UI itself never directly communicates with Kubernetes.

9.9.3 Test configuration

The administration panel can also be used to configure the types of test that may be run.



Language Classification

View and modify this test type.

Test name: Language Classification

Test tag: LANGANALYSIS

Test description: Detect the language of the text

Test result template (%RESULT% will be substituted with the result): %RESULT%

Highlight results? (Default)

Show results? (Default)

MODIFY TEST TYPE

Figure 14 Test configuration options

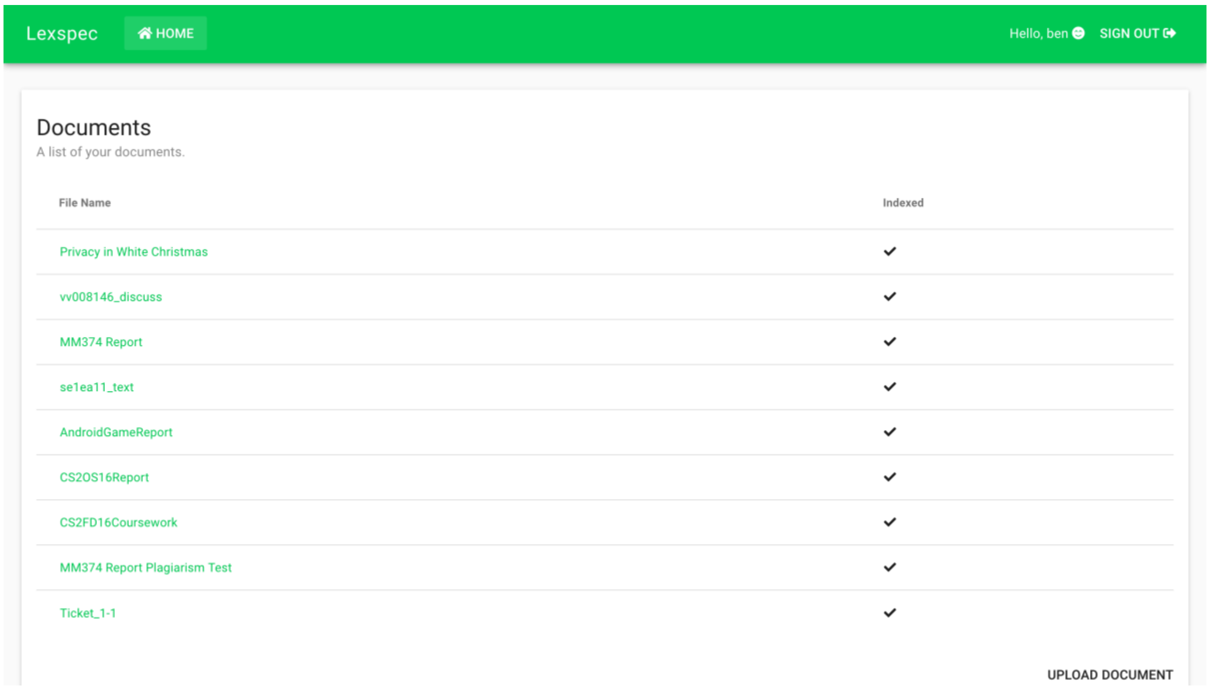
Figure 14 shows the options available. Note that configuring a test does not control when or how a test is run, it should instead be viewed as metadata that can be used by test services to classify the results they produce. Many of the options relate to how the test appears in both the administration panel and end-user interface, while a test's tag is used by test services to find a particular type of test.

There are some concerns around storing test configuration in this way – if the infrastructure needs to be removed and re-installed, the configuration details are lost as they are stored in a database. An alternate approach would be to adopt an "infrastructure as code" approach, where configuration values are expressed in files, often as YAML or JSON. These files are then read by the application to produce the desired configuration. One advantage of this approach is that the files themselves can be version controlled in order to support a more rigorous workflow around configuration changes. An implementation of this approach was outside the scope of this project, however.

Originally, the test type's ID in the database was used by test services to associate their results to a particular type of test, however it was found that this meant that if a test type was deleted and recreated for any reason (for example: if the platform was setup from scratch again), the results would no longer be correctly associated, and the test services themselves would require reconfiguration. The "test tag" field was implemented instead, where a test tag may remain the same across many different entries in a database over-time. While not as robust as the infrastructure-as-code approach outlined above, this does prevent from too many configuration changes needing to be made across the platform.

9.10 End-user interface

In comparison to the administration panel, any user is able to view the end-user interface. While it uses a separate API to the administration panel, the workflow is the same: an intermediate API sits between the UI displayed in the user's browser, and the microservices that power the platform.



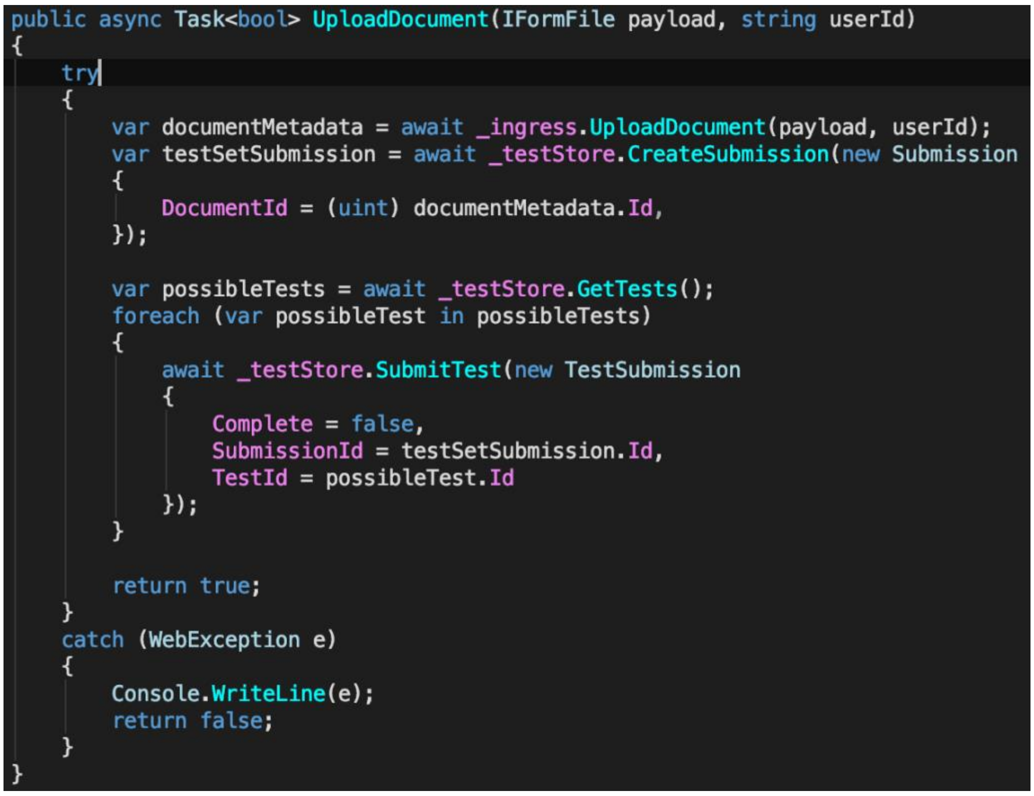
Documents

A list of your documents.

File Name	Indexed
Privacy in White Christmas	✓
vv008146_discuss	✓
MM374 Report	✓
se1ea11_text	✓
AndroidGameReport	✓
CS2OS16Report	✓
CS2FD16Coursework	✓
MM374 Report Plagiarism Test	✓
Ticket_1-1	✓

Figure 15 Lexspec's end-user interface

Figure 15 shows that while the end-user interface adopts a similar style to the administration panel, it is significantly simpler to use and contains far fewer options. A user is only able to view their own documents, not all documents across the platform. Notably, in its current implementation, a user cannot delete their own documents. This feature could be implemented with ease, or later implemented as a “toggle” where users could remove their own documents under specific circumstances.

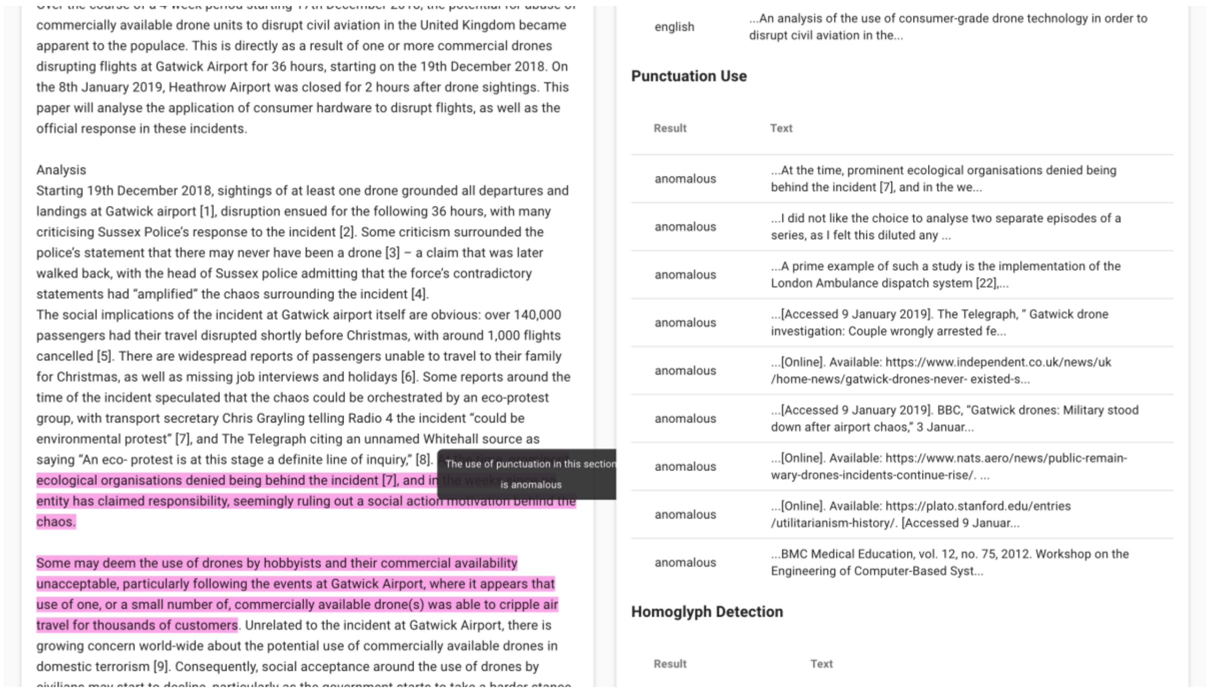


```
public async Task<bool> UploadDocument(IFormFile payload, string userId)
{
    try
    {
        var documentMetadata = await _ingress.UploadDocument(payload, userId);
        var testSetSubmission = await _testStore.CreateSubmission(new Submission
        {
            DocumentId = (uint) documentMetadata.Id,
        });

        var possibleTests = await _testStore.GetTests();
        foreach (var possibleTest in possibleTests)
        {
            await _testStore.SubmitTest(new TestSubmission
            {
                Complete = false,
                SubmissionId = testSetSubmission.Id,
                TestId = possibleTest.Id
            });
        }

        return true;
    }
    catch (WebException e)
    {
        Console.WriteLine(e);
        return false;
    }
}
```

Figure 16 Microservices being called to upload a document and submit tests

In order to simplify the platform's use for end-users, they are unable to manually run tests, or specifically choose which tests to run. Instead, when they upload a document, the intermediate API that sits between the UI and the platform's microservices automatically submits all the appropriate tests. The microservices approach to development makes this task very simple – the code to do so is displayed in figure 16.

Due to the well-defined APIs and communication routes established as a result of splitting the platform into a series of microservices, the appropriate services can be called to upload a document, and then submit all the necessary tests with very little boilerplate or overhead. This also abstracts away much of the complexity of the platform from individual services. In this case, the API for the front-end interface does not know anything about the RabbitMQ broker used to deliver test requests to individual test services, as this is handled exclusively by the test submission API. Had a monolith approach been taken, it may be that the frontend interface would need to deliver these messages to the RabbitMQ broker itself.


commercially available drone units to disrupt civil aviation in the United Kingdom became apparent to the populace. This is directly as a result of one or more commercial drones disrupting flights at Gatwick Airport for 36 hours, starting on the 19th December 2018. On the 8th January 2019, Heathrow Airport was closed for 2 hours after drone sightings. This paper will analyse the application of consumer hardware to disrupt flights, as well as the official response in these incidents.

Analysis
Starting 19th December 2018, sightings of at least one drone grounded all departures and landings at Gatwick airport [1], disruption ensued for the following 36 hours, with many criticising Sussex Police's response to the incident [2]. Some criticism surrounded the police's statement that there may never have been a drone [3] – a claim that was later walked back, with the head of Sussex police admitting that the force's contradictory statements had "amplified" the chaos surrounding the incident [4].
The social implications of the incident at Gatwick airport itself are obvious: over 140,000 passengers had their travel disrupted shortly before Christmas, with around 1,000 flights cancelled [5]. There are widespread reports of passengers unable to travel to their family for Christmas, as well as missing job interviews and holidays [6]. Some reports around the time of the incident speculated that the chaos could be orchestrated by an eco-protest group, with transport secretary Chris Grayling telling Radio 4 the incident "could be environmental protest" [7], and The Telegraph citing an unnamed Whitehall source as saying "An eco- protest is at this stage a definite line of inquiry," [8]. ecological organisations denied being behind the incident [7], and in the week entity has claimed responsibility, seemingly ruling out a social action motivation behind the chaos.

The use of punctuation in this section is anomalous

Some may deem the use of drones by hobbyists and their commercial availability unacceptable, particularly following the events at Gatwick Airport, where it appears that use of one, or a small number of, commercially available drone(s) was able to cripple air travel for thousands of customers. Unrelated to the incident at Gatwick Airport, there is growing concern world-wide about the potential use of commercially available drones in domestic terrorism [9]. Consequently, social acceptance around the use of drones by

english	...An analysis of the use of consumer-grade drone technology in order to disrupt civil aviation in the...

Punctuation Use

Result	Text
anomalous	...At the time, prominent ecological organisations denied being behind the incident [7], and in the we...
anomalous	...I did not like the choice to analyse two separate episodes of a series, as I felt this diluted any ...
anomalous	...A prime example of such a study is the implementation of the London Ambulance dispatch system [22],...
anomalous	...[Accessed 9 January 2019]. The Telegraph, " Gatwick drone investigation: Couple wrongly arrested fe...
anomalous	...[Online]. Available: https://www.independent.co.uk/news/uk /home-news/gatwick-drones-never- existed-s...
anomalous	...[Accessed 9 January 2019]. BBC, "Gatwick drones: Military stood down after airport chaos," 3 Januar...
anomalous	...[Online]. Available: https://www.nats.aero/news/public-remain-wary-drones-incidents-continue-rise/. ...
anomalous	...[Online]. Available: https://plato.stanford.edu/entries /utilitarianism-history/. [Accessed 9 Januar...
anomalous	...BMC Medical Education, vol. 12, no. 75, 2012. Workshop on the Engineering of Computer-Based Syst...

Homoglyph Detection

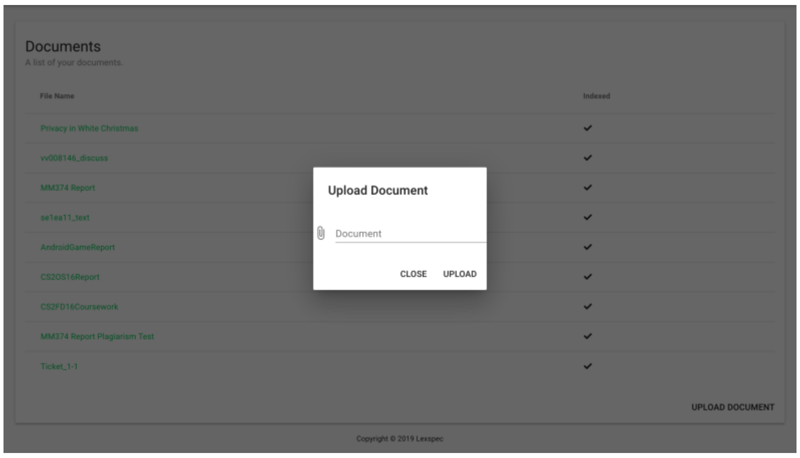
Result	Text

Figure 17 Report displayed in the end-user UI

As the user is unable to submit tests themselves, the end-user interface instead displays the results of the most recent test run when viewing a document's content. This is captured in figure 17, which shows how the results of the latest test submissions are listed on the right-hand side, with the appropriate sections of the document being highlighted. Each highlighted section is colour-coded to the test performed, and hovering over a highlighted section will display more information about why the section has been flagged as anomalous.

9.10.1 Usability

What the author wishes to convey in this section is that despite the overall complexity of the platform: numerous test services, content parsing services, a fully-featured authentication and authorisation scheme, as well as message brokers, multiple physical hosts and horizontally scalable APIs, the interface presented to the end-user is incredibly simple yet fully-featured.


Documents
A list of your documents.

File Name	Indexed
Privacy in White Christmas	✓
vv008146_discuss	✓
MM374 Report	✓
se1ea11_text	✓
AndroidGameReport	✓
CS2OS16Report	✓
CS2FD16Coursework	✓
MM374 Report Plagiarism Test	✓
Ticket_1-1	✓

Upload Document
Document
CLOSE UPLOAD

UPLOAD DOCUMENT

Copyright © 2019 Lexspec

Figure 18 Document upload modal

Figure 18 captures the steps the user must undertake in order to add a document to the platform: select a file in their file explorer, or drag-and-drop a file onto the "Document" field, and then click "Upload". The document's title and file type are detected automatically, and its content is parsed in the background with no input from the user, even if the original document is uploaded in a relatively complex format like a PDF.

Fully-featured operation by an end-user boils down to pressing 3 or 4 buttons. In order to support the author's belief that the platform is easy for an end-user to understand, several colleagues were consulted in an informal setting. None of these colleagues had any previous experience of the platform, but may have been familiar with other plagiarism detection tools. At the start of the session, they were given some general context: that Lexspec is a plagiarism detection platform, and it analyses the structure and features of text to detect suspicious passages. They were then provided with login credentials and asked to trial the application, and leave anonymous feedback. This feedback is presented below:

Participant	Feedback	Easy to use?
#1	Very intuitive to navigate. There could be better visual feedback that a document has been uploaded.	Yes
#2	Easy to use. Uncluttered	Yes
#3	Very clear, but I couldn't find a way to delete my document	Yes
#4	Easy to use. Can I upload the same document more than once and test it multiple times?	Yes
#5	Easy to use. Lots of false positives in my document, though	Yes

Overall, the results show that potential users generally find the platform simple to use with no prior training – although the author acknowledges that better visual hints could be displayed in some areas when a user performs an action. It's worth noting that these results may be skewed by the selection of participants, who are all generally very literate in technology, and also the small sample size. Had more time been available, a more comprehensive study of usability would have been undertaken.

10 Testing

Given the complex nature of the system, testing was vital to ensuring its correct operation. This is especially pertinent given the number of moving parts at work.

Where possible, all components of the system were unit tested. Specific care was given to unit test edge cases in critical services like the text mapping API, where any small modification to the existing code could break current uses of the API in unexpected ways.

10.1 Continuous Integration and Continuous Deployment

The use of a continuous integration and continuous deployment pipeline during the project's implementation meant that errors caused by an incorrect deployment of a new version can be kept to a minimum.

In most cases, new versions are deployed semi-automatically. The CI/CD platform detects that new changes have been uploaded to the source code repository. It then begins running the pipeline. For most projects, this involves compiling the code, running unit tests, and then waiting for a user to confirm whether or not to deploy the compiled code. If any step fails, the pipeline aborts.

In cases where a database schema change has taken place, a developer must then manually enter the container and execute the database migrations. This is usually done in one command, and only needs to be completed the first time the container starts after the update. It is likely that this process could be automated using sidecar containers, however given the rarity of a database schema change taking place it was decided this would be a premature optimisation.

A git branch model was used to ensure a stable production environment. Changes to the master branch of the repository are deployed to the production environment. Changes to any other branches ("feature branches") are deployed to the QA environment. By requiring a code review before code can be merged to the master branch (and therefore the production environment), poor quality code can be caught and rewritten to a higher standard, further reducing the likelihood of problematic software being deployed to a production environment.

10.2 End-to-end Testing

While unit testing ensures that each of a service's components work in isolation, it is also important to ensure that the system works as a single, cohesive mechanism. This is particularly important when the platform operates as a set of independent microservices, and consequently there are many moving parts which may fail or change, causing another service to malfunction.

The ideal end-to-end test would be to create a user account, upload a document, view the document, submit tests, and compare the test results against the expected results. Ideally this would be done through the UI, rather than the service APIs, as this is how a user would interact with the service.

If possible, all artefacts of the end-to-end test would be removed after the test, leaving a fresh slate for the next end-to-end test. By doing this, the results of a future test aren't affected by those of a previously run test, and any unintended dependencies between tests can be avoided.

The main form of end-to-end testing completed during the project's implementation was simply regularly testing different features of the platform as part of the development process. During this regular testing, several issues were identified. An example of such an issue is when the document metadata API was updated to include a new field, but the UI did not submit this field, resulting in attempts to upload a new document failing.

There were two main issues with this form of manual testing: tests weren't idempotent, and changes made as part of a test (for example: a document being uploaded) were not undone later. The testing was also completely manual, and relied entirely on the developer performing a variety of tests when implementing new features in the platform. While this is sufficient for development purposes, a production system should have some automated form of end-to-end testing that can be run at regular intervals, or in response to changes in the codebase.

10.3 QA Environment

In order to aid with testing and development, two environments were used throughout the implementation. The first environment was a QA environment, and all changes to the platform's codebase where deployed to this environment first. Once deployed here, a mix of end-to-end and UAT testing was completed. If the changes were deemed acceptable, and didn't introduce new bugs into the platform, then these changes were pushed to the production environment.

UAT testing was carried out by a small group of colleagues with no knowledge of the codebase or the internal workings of the platform. By carrying out this black box testing, testers had no expectations for how the platform should perform, and the testers' experience more closely reflected that of a genuine end-user.

Deployment to the production environment is handled entirely by the CI/CD pipeline. Any changes pushed to a repository's master branch are deployed to the platform's production environment. Meanwhile, any changes pushed to other branches ("feature branches") are deployed to the QA environment. Each environment is completely independent, using separate databases, services and user accounts. This means that any adverse effects encountered in the QA environment cannot impact the production environment.

10.4 Validation

In order to validate the performance of the platform, tests were performed on the final version to ensure it functioned as expected. The expectations and results of these tests are detailed below.

Test	Expected Result	Actual Result	Notes
Upload document through UI	User should be able to	User can	
Upload document through admin panel	Admin should be able to, user shouldn't	Admin can, user can't	
View other user's documents through UI	User shouldn't be able to view other users' documents	User can't view other users' documents	403 error
View all documents through admin panel	Admin should be able to view all documents	Admin can view all documents	
Tests run automatically when uploaded via UI	UI should automatically schedule tests on a new document, no input from user	Pass	
UI only shows latest test results	UI should only show user the latest tests results for a document	Pass	After running a new set of tests from the admin panel, only the latest results are displayed in the UI
Tests should be queued if no test service is available	If a test service is unavailable, requests for that test should be queued at the message broker	Pass	Tests are executed when a test service is available
Document parsing should be queued if no document parser available	If a document parser is unavailable, request should be queued until a parser is available	Pass	Document is parsed when document parser is available
User should be able to authenticate using Lexspec ID credentials	User should be able to authenticate using OIDC flows	Pass	OIDC implicit flow
Failed services should automatically restart	Failed services are restarted	Pass	Kubernetes reschedules failed services

11 Discussion

11.1 Contribution

Despite the limitations outlined below, overall the implementation of the project has been a success. An end-to-end platform has been implemented which is capable of supporting plagiarism detection services. Perhaps more importantly, the platform is implemented in a flexible manner which makes further development and expansion simple. This is particularly important in light of the shortcomings detailed later in the section.

In addition to the successful implementation of the platform, several marketable skills have been honed by the author, and competencies gained in several areas where skill was lacking. For example, knowledge of the OpenID Connect protocol and its implementations is a valuable skill within industry with the growing use of single sign-on services. Kubernetes is used at both small and large enterprises worldwide to manage workloads, and more widely Docker and container solutions are used as a software distribution mechanism in both enterprise and open-source projects. The author believes that these skills will prove bountiful in future endeavours, and that while developing a platform using these technologies was likely an increased workload compared to a monolith system, the increasing tendency of cloud-based applications to be developed in a distributed manner means that this effort has not been in vain.

Previous work experience in software development from a year in industry proved beneficial in the development of the platform. Best-practices were instilled around the areas of testing, writing self-documenting code, and distributing systems across multiple services. While the development of the platform likely would have been possible without this experience, the implementation itself has been significantly smoother than it otherwise may have been.

While the false-positive rate of anomalies detected is too high for general use, some key knowledge in this area has been gained, and results from other publications confirmed. Unsurprisingly, indicators of plagiarism first researched and published in academic journals, conferences, and other sources of a similar ilk have proved to be the most effective of all the implemented detection services. Less complex indicators (for example: solely examining the average word length) have been found to be too vague to be a practical signal in indicating potential plagiarism.

In particular, Norvig’s work on n-gram frequencies and distribution in the English language proved to be a valuable starting place in detecting anomalies in text. This, combined with Chow et al.’s work on using chi square tests to detect potential homoglyphs in text served as an excellent introduction to the topic.

On reflection, the amount of work required around text-processing and manipulation was underestimated at the beginning of the implementation and not accounted for when planning the implementation stages. Much of this work was required to format the ingested raw documents in a manner that was presentable on a web page. This formatting then meant that text indices required shifting to match the changed indices of characters in the text. Initial attempts to do this on an ad-hoc, per service basis proved to be inefficient, and eventually further time was committed to providing a dedicated service for this purpose.

In addition, the amount of time required to implement the apparatus surrounding the core test services was underestimated – this includes services like the document metadata datastore, test data store, and document ingress services. Additionally, how these services interact with one another could have been better planned prior to the implementation beginning.

The use of a QA environment and the Git VCS allowed for rapid development. A production environment was always available for testers to use, and using a VCS allowed for robust code changes and tracking – for example: it was easily possible to view the history of a file, or undo a certain change. These two factors, in combination with the use of a CI/CD pipeline made testing and deploying changes to the platform a simple and repeatable process. Furthermore, the CI/CD pipeline improved the robustness of the platform by stopping code changes from being deployed if tests failed against the new version. These best practices would also allow development on the project to be scaled from a single developer to a team of developers with ease.

11.1.1 Implementation Compared Against PID Objectives

This section will analyse the completed implementation against the initial specification laid out in the PID.

11.1.1.1 Third Party Data Sources

While a proof-of-concept was implemented in this area, in the form of a service which ingests random content from Wikipedia, it was determined that third-party data sources were not necessary for the core implementation of lexical anomaly detection. Not storing third-party content from other publishers also lessened the legal burden of the author.

Despite the limited implementation of this requirement, the proof-of-concept has sufficiently shown that the platform's ingress pipeline is flexible enough to handle the ingestion of content from a variety of sources. In the example of content ingested from Wikipedia, the article contents are sent in raw text format to the document ingress service. This lays the groundwork for other ingestion services (for example: from a news website) to operate in a similar fashion in the future, if required.

The container and microservice based architecture of the platform makes it simple to enable or disable specific services. In the case of the Wikipedia ingestion service, it is frequently disabled (zero instances running on the cluster), and can be enabled or disabled without affecting the rest of the platform, or requiring any other changes to the platform. Enabling or disabling the Wikipedia ingestion service is a case of issuing a single command, and could be automated in the future if required. The same principles would apply to any ingestion service – for example, in the future a Wikipedia ingestion service could be running, while a BBC News ingestion service is disabled.

11.1.1.2 Scalable Infrastructure for Data Storage and Analysis

This is probably the requirement that has been best satisfied. It is the author's belief that the system has been implemented in a flexible and scalable way, and this paves the way for any future development. The use of Elasticsearch to store document text allows content to be stored in an efficient way, while also providing free text searching. Additionally, Elasticsearch supports clustering, meaning that document content can be sharded across several physical hosts, and storage capacity can be horizontally scaled as demand grows. This provides a means to meet both data integrity and future expansion needs.

All relational databases are implemented using PostgreSQL, which can again be clustered. In addition, multiple clients can access a database simultaneously, compared to, say, SQLite, which can only handle a single client at a time, and performs poorly on networked filesystems.

By abstracting away underlying datastores from the rest of the platform, there is an opportunity for future development to move away from these datastores, or alternatively implement more intelligent mechanisms for accessing the underlying datastore. For example, this would enable a developer to later implement caching to return frequently queried test results or documents. It also means that a future developer isn't bound by today's decisions to use a specific data storage technology, and would be able to facilitate a managed migration to a new datastore.

By using message queues and microservices, document analysis services can be scaled horizontally as required by the demand on the platform, enabling faster processing of test requests. In any case, if the document analysis services become saturated with requests, future requests simply wait in a message queue until they can be processed at a later time. As it's not essential that a document be analysed immediately, this is an acceptable trade-off for the ability to scale horizontally, and wouldn't affect the platform in day-to-day usage.

Using a single and widely-supported container technology for the platform makes transitioning between hosting environments relatively easy – either to a new cloud provider, or to physical hardware nodes. The container orchestrator (Kubernetes) ensures that services are running, and also supports high availability behaviours, including restarting failed services, and distributing different instances of the same service across multiple physical hardware nodes to protect against hardware failure.

11.1.1.3 Application of Stylometric Analysis

As mentioned earlier in the section, Norvig's work on n-gram distribution and frequency in the English language has proved an invaluable starting point for lexical analysis. Further stylometric analysis was implemented around lexical features like word-length, average sentence length, frequency of punctuation marks, and more.

Based on the number of test services implemented that analyse documents using stylometry techniques, the author believes this requirement has been satisfied, and additional techniques could be implemented with ease at a later date, as research in this area advances.

11.1.1.4 Application of Machine Learning to Improve Accuracy

While the platform as a whole has a high false-positive rate, machine learning has been used in certain areas to improve the accuracy of some analysis services. In particular, the service which classifies a document's language based on its n-grams benefitted from using machine learning to find an acceptable range of unigram and bigram frequencies for the English language.

While this service originally used the values estimated by Norvig, it was found that these values were too brittle for real-world use analysing a variety of documents. It's hypothesised that the frequency of unigrams and bigrams may differ depending on the style of writing: for example, for a medical journal, versus for a tabloid newspaper. In order to improve the accuracy of the language classification system, an ensemble classifier was used to find a range of acceptable values for unigram and bigram frequency in the English language. Once this model was implemented in the classification service, the results were much more accurate.

Based on the analysis of the use of ensemble classifiers above, the author believes this requirement has been satisfied.

11.1.1.5 Pre/post-processing of Submitted Documents

This requirement is satisfied as part of the platform's document ingress pipeline. While it is simple to upload a document to a HTTP server, this is of little use to the platform unless its contents can be accurately parsed and stored. In some cases, for example when using plaintext files, this is a simple task. In other cases – PDF files, for example, this is a more complex undertaking.

Pre-processing is done on all documents that are uploaded to the platform. Their file type is identified using its extension (for example: .pdf), and then the document is passed to the appropriate parsing service so that the content can be extracted and stored in a plaintext format. Post-processing is undertaken when a document is presented to the user in the web UI. This involves formatting the document so that it is correctly displayed in the user's browser – for example: replacing new-line characters with the HTML break tag.

A number of services exist around the pre and post processing of documents as part of their ingress into the platform, and presentation to the user. Based on this information, the author believes that the platform does meet the requirement of performing pre and post processing on submitted documents.

11.1.2 Test Services Limitations

While the platform shows promise in several areas, there are several limitations in the current implementation which would prevent it from being used in practice. With further study and expertise in the fields of data mining and machine learning, the author believes that these limitations could be overcome.

The first limitation of the project has been the amount of data available for training. Despite exhaustive search efforts, there does not appear to be a publicly available repository of academic reports – particularly reports submitted by users. There are some providers of academic papers which could be accessed when required, but these documents don't satisfy the requirements of the project. Ideally, a large repository of reports would be available, categorised by author and submission date.

The reason for this is that many of the test services use an author's previous documents as training data to determine the veracity of the current document. During development, the only data available were previous reports from the project author, and previous reports of a small number of colleagues.

This lack of appropriate data during development leads to a more profound issue: there are doubts as to whether the average student would produce enough content across the course of their degree programme to generate sufficient training data for accurate plagiarism detection. Building on this, even if a sufficient quantity of data is generated over the course of a degree programme, the quality of the service provided is likely to be poor for a number of submissions until the quantity of training data increases.

Several approaches could be taken in order to tackle the training data quantity problem. Firstly, unsupervised learning methods could be investigated. While current test services heavily rely on previous submissions, if an unsupervised algorithm could be devised that relies solely on the content of the current document, the quality of the service provided would be less dependent on the quantity of training data. While unsupervised techniques exist for classifying text in an unsupervised manner – for example: using TFIDF as a feature, and K-Means clustering to classify, the author is unclear if such methods are available for author verification, or their effectiveness.

In addition to decreasing the dependence on training data quantity, investigations could be made into mining features with a stronger signal from the existing data. This would effectively allow the same extrapolations to be made from a smaller amount of data. More could also be done to prepare the data for analysis – currently no scaling or other preparation is carried out on features once they have been extracted from text, however in the case of several classifiers scaling the input data has been shown to improve the accuracy of the model [29].

Questions also exist around the quality and authenticity of training data. If a user consistently submits documents from a variety of authors (for example: if they regularly use an essay factory website) then the platform will struggle to identify consistent patterns in the author's writing style, leading to less accurate results. A similar, and more common, scenario would be an author who participates in a large number of group reports: the features from these group reports with several authors would skew the predictions of future documents, and the platform currently has no way of accounting for this.

Given ample time, the author would test implementing a meta-classifier: this classifier would take the results of other test services, and then determine whether a section of text is plagiarised. This would effectively work in a similar fashion to existing ensemble classifiers, although the exact implementation details have not been established.

11.1.3 Software Testing Limitations

The testing section above presented an overview of the testing strategies employed throughout the implementation. With further reflection, it seems that automated testing strategies would be required in order to properly scale the system beyond its current footprint, particularly if it were to be used in a production environment. The chief reasoning for this belief is that as the number of services grows, the interactions between these services also increase, making manual testing harder to execute and less effective. A full suite of end-to-end tests executed as part of a CI/CD pipeline would effectively eliminate regressions being pushed to a production instance.

A particular deficiency in the platform's testing capabilities is automated testing for regression in the text analysis services. The ideal implementation would be triggered whenever a test service was updated, or a new test service deployed. This regression tester would submit a document with known instances of plagiarism, and analyse the results returned by the test services in order to identify instances where plagiarism detection actually deteriorates as new services are released. This would allow a developer to either fix an implementation bug in the newly deployed service, or rollback to a release with better plagiarism detection coverage. This kind of functional testing is a more advanced testing case, but would likely prove to be an effective tool in practice. The author of this report has seen similar systems implemented in businesses which rely on predictive technology.

The unit test coverage of particular platform components is also varied. Some utility services with relatively complex logic are extremely well tested, with test cases covering both normal operation, edge cases, and bug fixes encountered during development. This is necessary to ensure that these utility services provide consistent information to their callers. On the other hand, several CRUD (Create, Read, Update, Delete) services – particularly those which sit in-front of databases, but have no other function) have very little coverage. This is in part due to their simple nature, but it would be preferable to have these simple cases tested. In particular, test cases should be devised for failure situations in these services: for example, if a malformed input is given.

One additional testing approach that could be implemented for increased confidence is killing production instances of services. Given the container-based nature of the platform, this should not affect the end-user's experience of the platform. Such tests are carried out by large technology companies – an example of which is Netflix, whose Chaos Monkey tool [30] randomly kills virtual machines in production. Netflix believes that regularly exposing engineers to production failures encourages them to build more resilient services [31]. This new discipline is commonly referred to as "Chaos Engineering", and revolves around artificially injecting or simulating failures in a service's production environment in order to ensure that the service can continue operating normally. By regularly exposing production services to failures, a developer's tendency to only test the "happy path" of their application can be avoided. This behaviour was observed and documented by professors at Virginia Tech, who found that while their students were achieving branch coverage of 95% in their tests, these same tests only detected an average of 13% of faults. Further analysis showed that students were testing the application's "happy path", rather than testing edge cases likely to introduce bugs to the application [32].

12 Social, Legal, Ethical, Health & Safety Issues

The project initiation document outlined the primary legal and ethical concerns around the platform. Namely, the storage and use of third-party content for analysis. Some of these concerns have been mitigated: no large library of external content has been developed, therefore any licensing concerns around storage and use of third-party content (eg: from news articles) has been avoided. A small amount of content has been ingested from Wikipedia, however Wikipedia's text is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License [33]. This allows the text to be used for any purpose (including commercial purposes) provided that the content is properly attributed.

In its current form, the ingested Wikipedia content is not used on the platform, however if it were to be in the future then attribution could be correctly given as a function of the document metadata datastore. During the development of the platform, explicit consent was granted from colleagues for the storage and use of their academic reports, and in the case that the platform were to be moved to production or commercial use, consent to the use and storage of a user's documents would be a condition of use of the platform.

Some thought has been given to GDPR since the project initiation document was completed. While the terms of use of the platform may necessitate that the user consents to their documents being stored and used, the platform must provide some way to remove this content if it's no longer required, or if the user wishes to remove their data from the platform. While this specific functionality hasn't yet been implemented, all of the groundwork has been laid. Every document is attributed to a user, and all data related to that document is appropriately attributed to it. The figure below outlines the process required if a user were to request the deletion of their data from the platform:

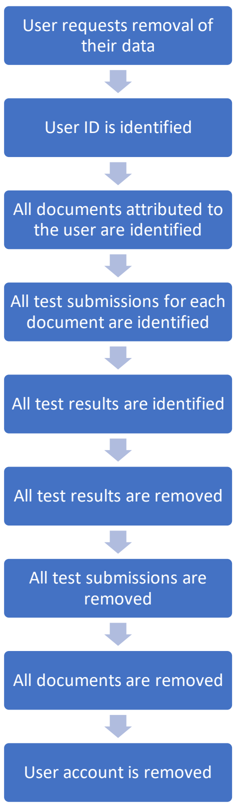

User requests removal of their data
User ID is identified
All documents attributed to the user are identified
All test submissions for each document are identified
All test results are identified
All test results are removed
All test submissions are removed
All documents are removed
User account is removed

Figure 19

While the removal of user accounts currently isn’t possible, a utility service has been written to support the deletion of a document from the platform. The service ensures that the document is not only deleted from the document store itself, but also its metadata entry and any associated test results are removed. With this in mind, the only work necessary to implement a feature to remove all of a user’s data from the platform is to iterate over all of a user’s documents and call this service.

No health and safety issues were identified as part of the project initiation document, and no issues have since been identified. The only social issue was whether it would be wise to ensure the platform returns accurate results, given the impact that a plagiarism detection service can have on a user’s career. With this in mind, it has been decided not to move the platform to production use, given the number of false positive results generated for plagiarism by the platform in its current iteration.

13 Conclusion

To conclude, a robust plagiarism detection platform has been implemented and deployed, with tooling setup to enable future development and repeatable deployments.

Referring back to the project initiation document: “The overall objective of the project is to produce a platform that can provide an end-to-end system for analysing documents for instances of plagiarised works, and produces a report highlighting instances of suspicious phrases.”. As detailed in the report above, this objective has been met, with all of the major requirements implemented in a working fashion.

A requirement that has not been fully implemented is ingestion of data from third-party sources, but this is largely because it was decided that this data was not required for the primary purpose of the platform: to detect lexical anomalies in an author’s work. Should the platform be expanded later to include fuzzy keyword matching, or any similar technique that compares a piece of work to an existing corpus, then it’s likely that this feature would be fully implemented. A proof-of-concept was implemented in this area, with content being ingested from Wikipedia, albeit with a few difficulties.

The primary problem when considering feature implementation is the accuracy of anomaly detections. While the features specification states “accurate detection” of different types of anomalies, at this stage the detection is not as accurate as hoped, although it is functional. It’s believed with future work the accuracy rate could be improved by building on the work done during the project’s implementation.

13.1 Future Improvements

Many of the future improvements outlined in various prior sections revolve around improving test accuracy. That’s not to say these are the only possible improvements, though. Reporting features could be improved. Currently, no notification system exists within the platform (for example: to notify of submission, or to notify when all tests have been completed). Such features could improve the user’s experience of the platform.

In order to better compete with fully-featured solutions like TurnItIn, it should be possible to setup “cohorts” – assign a leader and members of a cohort (usually “teacher” and “students”), where a cohort leader is able to create specific assignments to be submitted. Currently only the user who submitted a document is able to view it, and the test services implemented will perform poorly if documents written by multiple authors are submitted by the same user. In the envisaged solution, a cohort leader will be able to view documents submitted by any member of the cohort, including test results.

14 References

[1] OpenID Foundation, "OpenID Foundation," 2018. [Online]. Available: https://openid.net/foundation/. [Accessed 15 April 2019].

[2] OAuth, "OAuth 2.0," 2018. [Online]. Available: https://oauth.net/2/. [Accessed 16 April 2019].

[3] P. e. a. Martin, "Overview of the 4th International Competition on Plagiarism Detection," 2012. [Online]. Available: https://pdfs.semanticscholar.org/e287/607a13d7353cfd76da62d743b8a8759214f3.pdf. [Accessed 15 January 2019].

[4] D. e. a. Weber-Wulff, "Plagiarism Detection Software Test 2013," 2013. [Online]. Available: http://plagiat.htw-berlin.de/wp-content/uploads/Testbericht-2013-color.pdf. [Accessed 15 January 2019].

[5] BBC, "When essays for sale become contract cheating," BBC, 14 November 2012. [Online]. Available: https://www.bbc.co.uk/news/education-20298237. [Accessed 15 January 2019].

[6] BBC, "Cheating university students face FBI-style crackdown," BBC, 14 December 2018. [Online]. Available: https://www.bbc.co.uk/news/education-46530639. [Accessed 15 January 2019].

[7] W. B. Cavnar and J. M. Trenkle, "N-Gram-Based Text Categorization," in *SDAIR-94, 3rd Annual Symposium on Document Analysis and Information Retrieval*, Las Vegas, 1994.

[8] Y.-W. Chow, W. Susilo, I. Pranta and B. M. Ari, "Detecting visual spoofing using classical cryptanalysis methods in plagiarism detection systems," in *Applied Informatics and Technology Innovation Conference (AITIC)*, 2016.

[9] Urkund, "About Urkund," 2018. [Online]. Available: https://www.urkund.com/about-urkund/.

[10] TurnItIn LLC, "TurnItIn for Universities," 2018. [Online]. Available: https://www.turnitin.com/regions/uk/university.

[11] A. M. El Tahir Ali, H. M. Dahwa Abdulla and V. Snasel, "Overview and Comparison of Plagiarism Detection Tools," 13 5 2011. [Online]. Available: http://ceur-ws.org/Vol-706/poster22.pdf.

[12] Indigo Stream Technologies, "Copyscape Premium," 2018. [Online]. Available: https://www.copyscape.com/premium.php.

[13] P. Norvig, " English Letter Frequency Counts: Mayzner Revisited or ETAOIN SRHLDCU," 2013. [Online]. Available: norvig.com/mayzner.html. [Accessed 9 April 2019].

[14] M. S. Mayzner and M. E. Tresselt, "Tables of single-letter and digram frequency counts for various word-length and letter-position combinations," *Psychonomic Monograph Supplements,* pp. 13-32, 1965.

[15] Neal, Tempestt et al., " Surveying Stylometry Techniques and Applications," *ACM Computing Surveys (CSUR),* vol. 50, no. 6, pp. 86-122, 2018.

[16] F. B. Schneider, "Least Privilege and More," Cornell University, Ithaca.

[17] OpenID Foundation, "OpenID Certification," 1 April 2019. [Online]. Available: https://openid.net/certification/. [Accessed 9 April 2019].

[18] IdentityServer, "Using EntityFramework Core for configuration and operational data," 30 December 2018. [Online]. Available: http://docs.identityserver.io/en/latest/quickstarts/7_entity_framework.html. [Accessed 9 April 2019].

[19] Edureka, "Golang vs Python: Which One To Choose?," 15 October 2018. [Online]. Available: https://www.edureka.co/blog/golang-vs-python/. [Accessed 9 April 2019].

[20] Monzo, "Building a Modern Bank Backend," 19 September 2016. [Online]. Available: https://monzo.com/blog/2016/09/19/building-a-modern-bank-backend/. [Accessed 9 April 2019].

[21] T. Stanisz, J. Kwapień and S. Drożdż, "Linguistic data mining with complex networks:a stylometric-oriented approach," *arXiv,* 2019.

[22] W. F. Friedman, "The index of coincidence and its applications in cryptanalysis," Aegean ParkPress, 1987.

[23] CORE, "About CORE," The Open University, 21 January 2019. [Online]. Available: https://core.ac.uk/about/. [Accessed 9 April 2019].

[24] Pedregosa et al., "Scikit-learn: Machine Learning in Python," *Journal of Machine Learning Research,* vol. 12, pp. 2825-2830, 2011.

[25] scikit-learn, "Ensemble methods," scikit-learn, 1 March 2019. [Online]. Available: https://scikit-learn.org/stable/modules/ensemble.html. [Accessed 4 April 2019].

[26] E. You, "The Progressive JavaScript Framework," 2019. [Online]. Available: https://vuejs.org/. [Accessed 15 April 2019].

[27] The Linux Foundation, "Production-Grade Container Orchestration - Kubernetes," The Linux Foundation, 2019. [Online]. Available: https://kubernetes.io/. [Accessed 16 April 2019].

[28] Hashicorp, "Vault by Hashicorp," Hashicorp, 2019. [Online]. Available: https://www.vaultproject.io/. [Accessed 16 April 2019].

[29] scikit-learn, "Preprocessing Data," 2018. [Online]. Available: https://scikit-learn.org/stable/modules/preprocessing.html. [Accessed 24 April 2019].

[30] Netflix, "Chaos Monkey," Netflix, 24 February 2019. [Online]. Available: https://github.com/netflix/chaosmonkey. [Accessed 9 April 2019].

[31] Basiri, Ali; Behnam, Niosha; de Rooij, Ruud, et al., “Chaos Engineering,” *IEEE Software,* vol. 33, no. 3, pp. 35-41, 2016.

[32] S. H. Edwards and Z. Shams, “Do Student Programmers All Tend to Write the Same Software Tests?,” in *Innovation and Technology in Computer Science Education*, Uppsala, Sweden, 2014.

[33] Wikipedia, “Wikipedia:Reusing Wikipedia content,” Wikipedia, 31 October 2017. [Online]. Available: https://en.wikipedia.org/wiki/Wikipedia:Reusing_Wikipedia_content#Re-use_of_text_under_Creative_Commons_Attribution-ShareAlike. [Accessed 9 April 2019].

[34] Academic Paradigms LLC, “Service Features,” 2018. [Online]. Available: https://www.checkforplagiarism.net/service-features/features. [Accessed 15 January 2019].

15 Appendices

Individual Project (CS3IP16)

Department of Computer Science
University of Reading

Project Initiation Document

PID Sign-Off

Student No.	**24008146**
Student Name	**Benjamin Magee**
Email	**vv008146@reading.ac.uk**
Degree programme (BSc CS/BSc IT)	**BSc CS**
Supervisor Name	**Varun Ojha**
Supervisor Signature	
Date	**01/10/2018**

SECTION 1 – General Information

Project Identification

1.1	**Project ID** (as in handbook)
	OWN
1.2	**Project Title**
	Implementing modern machine learning and data mining techniques to identify instances of plagiarism and fraudulent behaviour in published works
1.3	**Briefly describe the main purpose of the project in no more than 25 words**
	To identify plagiarism through stylometric analysis and fuzzy phrase recognition against a repository of works, providing more insightful and intelligent plagiarism detection than existing solutions.

Student Identification

1.4	**Student Name(s), Course, Email address(s)** e.g. Anne Other, BSc CS, a.other@student.reading.ac.uk
	Benjamin Magee, BSc CS, vv008146@reading.ac.uk

Supervisor Identification

1.5	**Primary Supervisor Name, Email address** e.g. Prof Anne Other, a.other@reading.ac.uk
	Varun Ojha, v.k.ojha@reading.ac.uk
1.6	**Secondary Supervisor Name, Email address** Only fill in this section if a secondary supervisor has been assigned to your project

Company Partner (only complete if there is a company involved)

1.7	**Company Name**
1.8	**Company Address**

1.9	**Name, email and phone number of Company Supervisor or Primary Contact**

SECTION 2 – Project Description

2.1 **Summarise the background research for the project in about 400 words. You must include references in this section but don't count them in the word count.**

Existing solutions in this space primarily focus on matching text similarity (Turnitin, 2013). While this provides adequate matching of phrases lifted directly from known works, it will not detect instances of plagiarism where the original text has been sufficiently modified in an attempt to avoid such detection methods.

A lesser explored space is the concept of using stylometry techniques in order to identify sections of a document where the writing style and content is out-of-place. Key metrics such as word length, complexity, use of grammatical structures, and other indicators of authorship are used to identify passages that are likely not original work, even if the original source is unknown and has not been indexed previously by the system.

Stylometric analysis can be used across a single document, or a "profile" of a given author can be built up over time, using a series of submitted works, to give a more accurate understanding of the author's writing style.

It is the author's belief that combining stylometry analysis with fuzzy keyword matching will provide a more accurate and insightful plagiarism detection platform. Stylometry techniques have been used for centuries to authenticate authorship, with examples ranging from Bible books authorship to the US Federalist papers (Jenkins & Jacob Jenkins, 2014). The same techniques can be applied in a new domain to ensure that authored content is original. Existing solutions already use fuzzy text matching with apparent success - commercial solutions such as Quetext use this technique on their platform (Quetext, n.d.) with good reviews, so there is no reason to doubt that combining two existing techniques won't be successful in its own right.

The final proposed component of a comprehensive and insightful plagiarism detection platform is identifying correctly referenced phrases from a bibliography using pattern matching - probably through regular expressions. This is possible because references must follow a defined format, for example the author-date system that Harvard referencing uses, and this should allow parsing of a bibliography, and applying simple phrase detection in the parsed document (if indexed), to eliminate false positives arising from correctly referenced works.

2.2 **Summarise the project objectives and outputs in about 400 words.**
These objectives and outputs should appear as tasks, milestones and deliverables in your project plan. In general, an objective is something you can do and an output is something you produce – one leads to the other.

The overall objective of the project is to produce a platform that can provide an end-to-end system for analysing documents for instances of plagiarised works, and produces a report highlighting instances of suspicious phrases.

The first objective will be to parse text from existing sources. This could include public domain academic works, public domain books, and content from websites (such as Wikipedia). This will act as an inital corpus of works from which keyword and phrase recognition can be performed. The output of this objective will be a searchable index of these documents, stored in ElasticSearch, a database system that supports a range of search operations, including good support for full-text search.

The document store must be scalable. While the initial dataset that acts as a corpus for phrase comparison will be relatively small, it must be able to scale efficiently as this dataset grows. To this end, a secondary objective is proposed: the dataset should be stored in a medium that supports growth, with an outcome of a responsive and workable dataset at any size. If ElasticSearch is chosen as a datastore, it has support for sharding and horizontal scaling (Elasticsearch, n.d.).

Once a document store is established, the next objective will be to provide a way of ingesting user-submitted documents. The final deliverable will be a system that accepts user input from a web form, and stores parsed text. There will be several smaller objectives to achieve this goal: notably around supporting multiple input formats, and normalising them to a single output format (likely plain text), that is indexed into the document store mentioned above. This system will likely be written in a language with first-class support for web services: PHP or C# are likely candidates.

Once documents can be submitted, the next milestone will involve the bulk of the plagiarism detection work. A system should be developed that is able to take a submitted document, and analyse it for instances of plagiarised work using the techniques mentioned in the research section. The results of this analysis should then be stored in a SQL database, linking the generated metadata to a specific document with a document ID that's common across all datastores.

Once plagiarism analysis is complete, the results should be presented in an easy-to-consume format. Likely a report of some kind. This report will be made available through a web portal. Given that the target sector will be education, it is likely the final report will need to be available to both the submitter, and some form of course coordinator. The report generation and presentation may be split across two systems using multiple tools. This allows for better decoupling of business logic from presentation.

The final system present a user-friendly web portal for document submission and results presentation, backed by a microservices architecture. Using a microservices architecture allows the problem to be split into its component domains, and enables the development of a flexible and testable system.

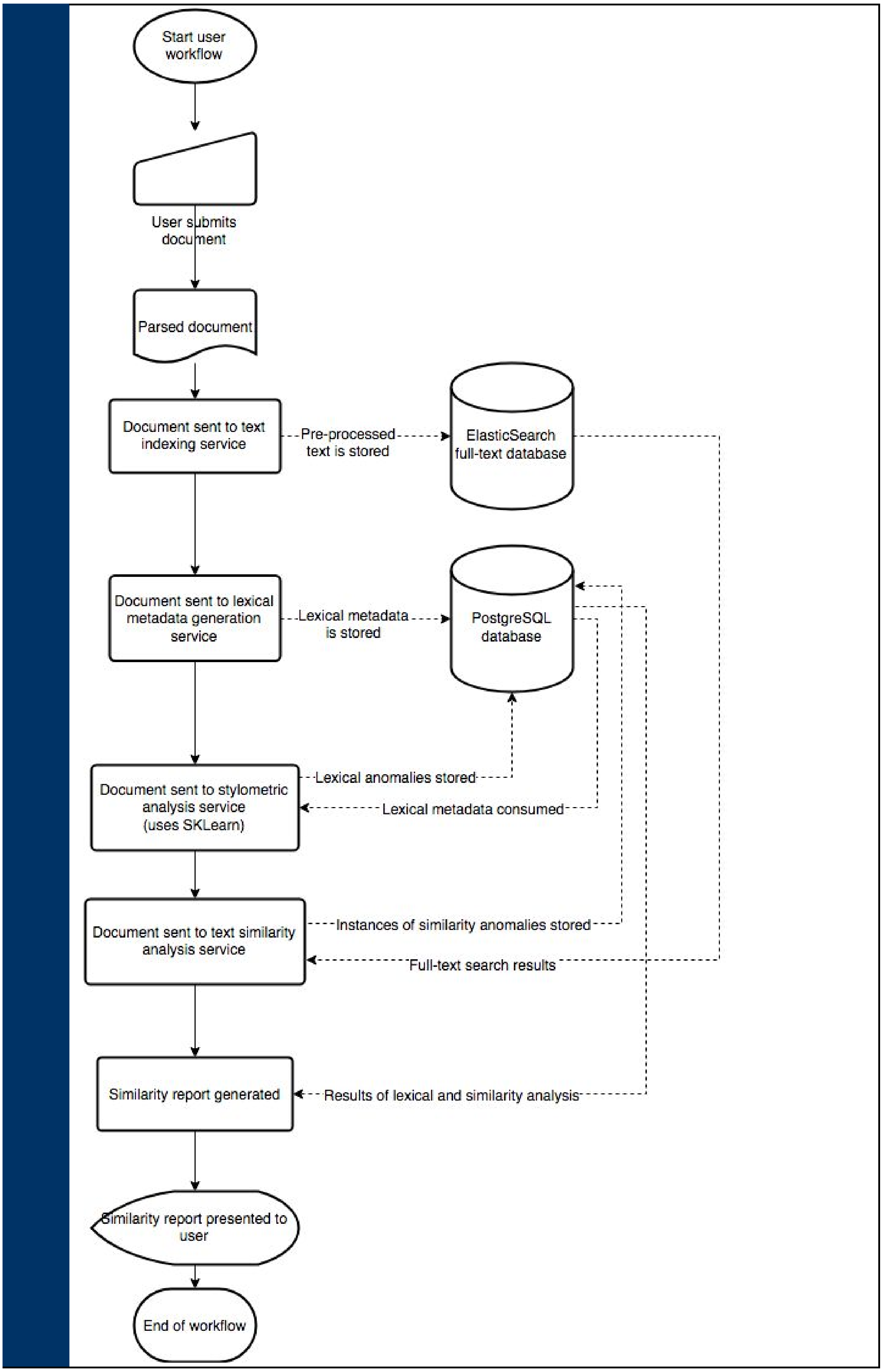
2.3	**Initial project specification - list key features and functions of your finished project.** Remember that a specification should not usually propose the solution. For example, your project may require open source datasets so add that to the specification but don't state how that data-link will be achieved – that comes later.

Requirements

- Numerous third party data sources
    - Newspapers
    - Wikipedia
    - Public domain texts
- Scalable infrastructure for data storage and analysis
- Application of stylometric analysis
- Application of machine learning to improve accuracy
- Pre/post-processing of submitted documents

Features / functions

- Accurate detection of out-of-character phrases
- Accurate detection of text lifted (or heavily borrowed) from another source with accredition
- Relatively low rate of false positives thanks to parsing of bibliographies (where appropriate)
- Exclusion of common phrases for lower detection rate
- Reporting features based on candidate, or body of work


Start user workflow
User submits document
Parsed document
Document sent to text indexing service
Pre-processed text is stored
ElasticSearch full-text database
Document sent to lexical metadata generation service
Lexical metadata is stored
PostgreSQL database
Document sent to stylometric analysis service (uses SKLearn)
Lexical anomalies stored
Lexical metadata consumed
Document sent to text similarity analysis service
Instances of similarity anomalies stored
Full-text search results
Similarity report generated
Results of lexical and similarity analysis
Similarity report presented to user
End of workflow

2.4	**Describe the social, legal and ethical issues that apply to your project. Does your project require ethical approval? (If your project requires a questionnaire/interview for conducting research and/or collecting data, you will need to apply for an ethical approval)**
	Primary legal and ethical issues may arise from the use of 3rd party data for analysis. This could easily be worked around, by using data from the public domain, or with the consent of the author. In order to improve the models used by the data, submitted documents would be stored for referencing later. This could be a condition of using the system. Some sensitive data (for example: name, institution) could be removed using pattern matching. There are no major social issues relating to the detection of plagiarism. It may be considered a social and/or ethical duty to ensure the accuracy of the tool if its reports will have real-life impacts on students and authors.
2.5	**Identify and lists the items you expect to need to purchase for your project. Specify the cost (include VAT and shipping if known) of each item as well as the supplier.** e.g. item 1 name, supplier, cost
	● Google Kubernetes Engine time for hosting of microservices developed, scalable infrastructure in cases where large computation required. Other providers could be used, pricing dependent on resources used, and time allocated.
2.6	**State whether you need access to specific resources within the department or the University e.g. special devices and workshop**
	● Anonymised past essay/assignment submissions would be beneficial in building a datastore and training model for predicting instances of plagiarised content.

8

SECTION 3 – Project Plan

3.1	**Project Plan** Split your project work into sections/categories/phases and add tasks for each of these sections. It is likely that the high-level objectives you identified in section 2.2 become sections here. The outputs from section 2.2 should appear in the Outputs column here. Remember to include tasks for your project presentation, project demos, producing your poster, and writing up your report.

Task No.	Task description	Effort (weeks)	Outputs
1	**Background Research**	3	
1.1	Existing solutions	1	Analysis of existing solutions and techniques applied, shortcomings.
1.2	Applications of stylometry analysis	1	How to apply stylometry analysis, existing reference implementations, key metrics to analyse
1.3	Applications of machine learning	1	How to use machine learning to improve accuracy of stylometric analysis
2	**Analysis and design**	3	
2.1	Identification of datastores	1	UML (or other) diagram detailing datastores to use
2.2	Identification of business use	1	Archimate diagram different system levels (business, application, IT)
2.3	Identification of problem domains	1	Distinct list of microservices and their problem domains, diagram of interactions
3	**Develop prototype**	9	
3.1	Crawl and store existing documents	1	Service(s) that crawl existing content from public domain sources and store it
3.2	Authentication	1	Service(s) to support authentication and attribution of resources.
3.3	Allow submission of new documents	2	UI and backend service(s) that allow submission of documents for analysis
3.4	Analyse submitted documents for stylometric anomalies and lifted text	3	Service(s) that store output of analysis on document in a parseable format
3.5	Generate report of analysis results	2	Generate report of analysis results and present it to the user.
4	**Testing, evaluation/validation**		
4.1	unit testing	1	Unit testing should take place as part of the development process, but

			some time will be alloted to tidy up testing
4.2	End-user evaluation	1	Feedback from end-users on accuracy and ease-of-use of system
4.3	Ongoing end-to-end/validation testing	1	Detection of “honeypot” documents submitted to the system on a regular basis
5	**Assessments**		
5.1	write-up project report	4	Project Report
5.2	produce poster	0.5	Poster
5.3	Presentation preparation	1	Presentation
TOTAL	**Sum of total effort in weeks**	**23**	

SECTION 4 - Time Plan for the proposed Project work

For each task identified in 3.1, please *shade* the weeks when you'll be working on that task. You should also mark target
To shade a cell in MS Word, move the mouse to the top left of cell until the curser becomes an arrow pointing up, left cl
select 'borders and shading'. Under the shading tab pick an appropriate grey colour and click ok.

	START DATE: ../../.... **<enter the project start date here>** **Project Weeks**							
Project stage	0-3	3-6	6-9	9-12	12-15	15-18	18-21	21-24
1 Background Research								
Existing solutions	■							
Applications of stylometry analysis	■							
Applications of machine learning	■							
2 Analysis/Design								
Identification of datastores		■						
Identification of business uses		■						
Identification of problem domains		■						
3 Develop prototype.								
Crawl and store existing documents			■					
Authentication			■					
Allow submission of new documents			■	■				
Analyse submitted documents for stylometric anomalies and lifted text				■	■			
Generate report of analysis results					■			
4 Testing								
unit testing						■		
End-user evaluation						■		
Ongoing end-to-end/validation testing						■		
5 Assessments								
write-up project report							■	■
produce poster							■	
Presentation preparation							■	

RISK ASSESSMENT FORM

Assessment Reference No.	**1**	**Area or activity assessed:**	**Creating project**
Assessment date	**01/10/2018**		

Persons who may be affected by the activity (i.e. are at risk)	**Project owner**		

SECTION 1: Identify Hazards - *Consider the activity or work area and identify if any of the hazards listed below are significant (tick the boxes that apply).*

1.	Fall of person (from work at height)		**6.**	Lighting levels		**11.**	Use of portable tools / equipment		**16.**	Vehicles / driving at work		**21.**	Ha ch
2.	Fall of objects		**7.**	Heating & ventilation		**12.**	Fixed machinery or lifting equipment		**17.**	Outdoor work / extreme weather		**22.**	Ha bi
3.	Slips, Trips & Housekeepin g		**8.**	Layout , storage, space, obstructions		**13.**	Pressure vessels		**18.**	Fieldtrips / field work		**23.**	Co asp
4.	Manual handling operations		**9.**	Welfare facilities		**14.**	Noise or Vibration		**19.**	Radiation sources		**24.**	Co Bu
5.	Display screen equipment		**10.**	Electrical Equipment		**15.**	Fire hazards & flammable material		**20.**	Work with lasers		**25.**	Fo

SECTION 2: Risk Controls - *For each hazard identified in Section 1, complete Section 2.*

Hazard No.	Hazard Description	Existing controls to reduce risk	**Risk Level** (tick one)			Furthe
			High	Med	Low	*(provid*
Name of Assessor(s)			**SIGNED**			
Review date						

Health and Safety Risk Assessments – continuation sheet

Assessment Reference N	
Continuation sheet nur	

SECTION 2 continued: Risk Controls

Hazar d No.	Hazard Description	Existing controls to reduce risk	**Risk Level** (tick one)			Furt
			High	Med	Low	*(prov actio*

Name of Assessor(s)		**SIGNED**
Review date		

Elasticsearch, B. V. (n.d.). ElasticSearch: RESTful, Distributed Search & Analytics. Retrieved October 4, 2018, from https://www.elastic.co/products/elasticsearch

Jenkins, J. J., & Jacob Jenkins, J. (2014). Stylometry. In *Encyclopedia of Deception*.

Quetext. (n.d.). Deepsearch Technology. Retrieved October 2, 2018, from https://www.quetext.com/deepsearch-technology

Turnitin, L. L. C. (2013, May 23). Top 15 Misconceptions About Turnitin. Retrieved October 2, 2018, from https://www.turnitin.com/blog/top-15-misconceptions-about-turnitin

Lexspec Logbook

Saturday, 29/09/2018

lxid

- IdentityServer4 install
- Licence
- README
- IdentityServer configuration

Monday, 01/10/2018

lxid

- IDSrv4

Tuesday, 02/10/2018

lxid

- Out of the box UI

Wednesday, 03/10/2018

lxid

- Refactor quickstart UI into codebase
- PostgreSQL backed lxID
- Removed unused directory, added migrations

Thursday, 04/10/2018

lxid

- Set workdir for static files
- Correct authentication scheme
- Remove loopback address
- Rename secret name
- Remove probes
- Helm chart
- Dockerfile, Jenkinsfile
- Registration, lowercase routes

Friday, 05/10/2018

lxid

- Change environment name
- Configuration file

Thursday, 11/10/2018

lxingress

- Code against interface, with implementation supplied
- Ignore temp files
- Remove temp files
- PoC

Saturday, 13/10/2018

lxingress

- Implemented PoC file download. Still needs auth and DB lookup.

Friday, 19/10/2018

lxingress

- Re-organised document and FS code. Need to move tests to new format, too
- Renamed files

Monday, 29/10/2018

metadoc

- Move database credentials into k8s secret
- Quote port
- Quote port
- Changed image name
- Jenkinsfile
- Updated repository path£
- Postgres library must be built from source on Alpine
- Binary
- pg_config command

Tuesday, 30/10/2018

metadoc

- QA values
- Updated serializer
- Django requires allowed hosts, use the current IP as an allowed host as this pod could bounce around the cluster. If this isn't added Django won't be able to serve requests to cluster members (required for intra-cluster communication as well as liveness/ready checks)
- Add localhost to allowed_hosts. Will need to add cluster service address later.
- Document path

Thursday, 08/11/2018

lx-parser-pdf

- Import

Wednesday, 14/11/2018

lx-parser-pdf

- Updated production RabbitMQ values
- Dockerfile, Jenkinsfile
- Removed compiled python files
- Use class

metadoc

- QA changes
- Include branch name in tag
- Update QA release name as Helm release names are global, not per-namespace

---

Thursday, 15/11/2018

lxingress

- Download tests passing
- PoC and Docker, Jenkins setup
- Helm
- Added ingress
- Metadoc client tests
- Refactoring ingress into multiple files and tests, improving testability with pluggable functions

---

Sunday, 18/11/2018

lxingress

- Upload tests pass

Tuesday, 20/11/2018

lxingress

- Corrected comment
- Post to MQ
- RabbitMQ configuration into Helm
- Fixed syntax error
- Corrected values
- Added some more upload tests

metadoc

- Allow all hosts

Wednesday, 21/11/2018

metadoc

- Corrected ALLOWED_HOSTS

Friday, 23/11/2018

lx-parser-pdf

- Parse as string
- Manual RabbitMQ ack
- Parse paragraphs
- Receive documents
- Added test for bullet points
- Ignore PyCharm virtual environment
- Normalised RabbitMQ exchange declaration between lxparserpdf and lxingress
- lxingress connection
- Debug logging

lxingress

- Typed field for ID
- Typed created field

metadoc

- Serialize ID

Saturday, 24/11/2018

lx-parser-pdf

- RabbitMQ ack test, JSON decoding test

Thursday, 06/12/2018

lx-parser-pdf

- Rename document to be documentmetadata
- ElasticSearch wrapper definition

Friday, 07/12/2018

lx-parser-pdf

- Reduce logging
- Quote
- Tidying up formatting
- Submit document ID
- ElasticSearch library requirements
- Egress PoC
- Don't try and convert
- QA

Saturday, 08/12/2018

lx-parser-pdf

- Stop printing body of message, parse paragraphs instead of plain text

---

Friday, 14/12/2018

lxid

- Started updating config

Saturday, 15/12/2018

lxid

- Update password
- Configurable database connection string, with IDesignTimeDbContextFactory implementation.
- Configurable database, QA setup
- Updated dev value

Sunday, 16/12/2018

lxid

- List clients, related properties need to be eagerly loaded in the controller.
- Lexspec user class

Tuesday, 18/12/2018

lxid

- Created empty form
- Store selected claims in session
- Whitespace
- Tidied up menu
- Modify CSP to allow FontAwesome CDN, add icon to layout
- Align icon

Wednesday, 19/12/2018

lxid

- Migrations
- Only show grants link if logged in
- Always show splash page
- Updated splash page
- Updated context factories for IDsrv migrations to set the migrations assembly correctly, regenerated migrations
- Register link
- Whitespace
- Move menu items around
- Updated brand

Thursday, 20/12/2018

lxid

- More create client form work, seed database with standard identity resources
- Add new clients

Friday, 21/12/2018

lexinspector

- init

lxid

- Simple CORS setup, should define specific origins later.
- CORS again
- Browser access tokens

Saturday, 22/12/2018

lxid

- Delete clients
- Edit clients (WIP)
- Update clients
- Only update secrets if new secret(s) provided
- Add dummy model bindings to fix value displayed when client is created
- Middle-align table contents
- Fix bug for empty redirect URI fields

Friday, 04/01/2019

lxingress

- Updated rabbitmq credentials for QA
- Invoke kubewrapper

lexinspector

- Auth working
- Vue boilerplate

metadoc

- Invoke kubewrapper

lx-parser-pdf

- Invoke kubewrapper
- Updated rabbitmq password

lxid

- Use kubewrapper in Jenkinsfile
- Invoke kubewrapper for production releases, too

Saturday, 05/01/2019

lexinspector

- UI work
- Authentication
- Highlight active item
- ApexCharts

Sunday, 06/01/2019

lexinspector

- Move dummy chart into component
- Added cypress for e2e tests
- Move hidden file
- Move UI into own folder
- Chart example

lxid

Monday, 07/01/2019

lxid

- Add API scopes to a client
- CRUD ApiResources

Tuesday, 08/01/2019

lexinspector

- RBAC and redirects
- RBAC API side, corrected UI to print access token, not identity token twice.

lxid

- Add user claims to API access tokens

Thursday, 10/01/2019

lexinspector

- Updated gitignore
- Added sln

Friday, 11/01/2019

lexinspector

- Started loading some settings from a config file
- Single user manager instance
- Load chart data from API

Friday, 18/01/2019

lexinspector

- Made lexinspector API URL configurable
- Document list

lxid

Tuesday, 22/01/2019

lexinspector

- Removed subdirectory, added Jenkinsfile
- Dockerfile
- Change directory

Thursday, 24/01/2019

lexinspector

- Receive PEM bundle from Vault, convert to X509Certificate2 for use in C#. Began writing an integration test
- Removed unused password, add test project CSProj
- remove custom directory
- Moved API to root

- Disabled strict TS compilation due to missing types for vue-material and apex charts, added Docker image
- Helm chart
- Change probe path
- Updated comment
- Added health check endpoint
- Removed left-over folder
- Get all documents
- Certificate integration test
- Add vault dependency

lexinspectorui

- Moved UI to its own repo
- Jenkinsfile
- Remove probes
- Documents page
- Use table
- Drone CI test
- Table background color
- Set root directory
- Configure NGINX
- Include TypeScript locally, move callback URLs to config
- Updated to use new ingress format
- Ingress changes
- Values
- Generate client certs
- Helm
- NGINX config from https://router.vuejs.org/guide/essentials/history-mode.html#example-server-configurations
- use config url

Friday, 25/01/2019

lexinspectorui

- Platform status

lexinspector

- Kubernetes controller
- Refactored settings for Kubernetes and Vault
- NetCoreApp 2.2
- Require authorization
- Updated dockerfile for aspnet 2.2
- Shorter TTL as a new certificate is requested with each request
- Configurable kubernetes config, convert from PEM to X509 (certificate only, no private key. Used for k8s CA)

- Basic Kubernetes integration
- Formatting changes

Saturday, 26/01/2019

lx-parser-txt

- gitignore
- Basic body

Wednesday, 30/01/2019

lx-parser-txt

- Jenkins and Docker configuration
- Remove test as no service has been created
- removed ingress
- Removed notes

lexinspectorui

- Included txt parsing service

Thursday, 31/01/2019

docstore

- Updated .gitignore to include compiled files
- Basic PoC

Friday, 01/02/2019

docstore

- Configuration
- Helm chart
- Dockerized build
- Corrected path
- Remove unused fields
- Jenkinsfile
- QA values

metadoc

docerase

- Initial project

Sunday, 03/02/2019

docerase

- Staging config
- Working PoC
- Copy files to output directory

Wednesday, 06/02/2019

docerase

- Health checks
- Helm and Jenkins

lexinspector

- Update config
- Bump compatibility
- Remove UI
- Merge fix
- Delete documents
- Add IoC bindings

lexinspectorui

- Delete documents
- Upload document dialog
- Updated QA config

Saturday, 09/02/2019

lexinspector

- Upload document

Sunday, 10/02/2019

lexinspector

- Application configuration
- IoC binding

Tuesday, 12/02/2019

lxid

- Logo

lexinspectorui

- Upload documents

Thursday, 14/02/2019

docstore

- Update metadata with indexed setting

Monday, 18/02/2019

lx-parser-pdf

- print url
- Update config
- PDF parsing to docstore instead of elasticearch

Sunday, 24/02/2019

lx-parser-txt

- Use docstore
- Remove string to bool conversion
- Log text
- Seek in buffer
- Configuration changes

lx-test-store-api

- URL config
- Viewset
- Submission test serializer
- Ignore compiled files
- Django
- All viewsets.
- Requirements
- Ignore debug database
- Migrations and routes

lx-parser-pdf

- Log response
- JSON

lxingress

- Bind message to correct queue depending on file type

Monday, 25/02/2019

lx-test-store-api

- Corrected models and migrations
- Separate production and dev settings
- Don't run migrations
- Helm chart and Dockerfile

testsubmissionapi

- Copy config
- Started rabbitmq config
- Unused imports
- Helm
- Make method static
- CORS
- Removed valuescontroller
- Swagger
- Enable health checks
- Submit test
- Create submission
- Project!
- Staging configuration

Tuesday, 26/02/2019

lexinspectorui

- Added test services to UI

testsubmissionapi

- Publish submitted tests to RabbitMQ

Friday, 01/03/2019

languagealyzer

- Chi square PoC
- Initial Poc

Sunday, 03/03/2019

languagealyzer

- Refactored code into classes, exported environment requirements.

Monday, 04/03/2019

languagealyzer

- Map modified text to original text
- Text mapping
- Define constant for separating character
- Comment constant
- Fixed missing final character
- Removed some debugging prints

Tuesday, 05/03/2019

languagealyzer

- Quote port number
- Helm
- Helm template, Jenkinsfile, Dockerfile
- Load document
- Corrected image
- Listen on RabbitMQ exchange and post results to test store

Thursday, 07/03/2019

lexinspectorui

- Fixed graph, added language analysis service

Friday, 08/03/2019

lexinspector

- Added docstore client template

Saturday, 09/03/2019

lexinspector

- WIP
- Get document text, fixed timeline bug

lexinspectorui

- Added document ID to list of documents
- Display document content

Sunday, 10/03/2019

lexinspector

- Added regex to convert plaintext documents to HTML. Added test case for this.
- Updated HTML parsing
- Updated HTML parsing

lx-test-store-api

- List test submissions for a particular document

lexinspectorui

- HTML now generated server-side
- Attempt redirect if user undefined

Monday, 11/03/2019

lexinspectorui

- Redirect to login if access token has expired
- Removed redirect code that wasn't working
- Added test metadata page

lx-test-store-api

- Add more fields related to test metadata
- Migrations
- Updated serializers and viewsets
- Register URLs

Tuesday, 12/03/2019

lexinspectorui

- Show individual test types
- Update existing test type
- Loading indicators
- Test table metadata
- Test table formatting

lexinspector

- List tests
- Get single test information
- Update existing test

Wednesday, 13/03/2019

testsubmissionapi

- Update to use test ID

lexinspector

- Create new test type

lexinspectorui

- Limit types
- Added basic typings for loading overlay
- Create new test type
- Removed debugging logging

Thursday, 14/03/2019

lexinspectorui

- Move documents list into card
- Moved documents to using nested routes

languagealyzer

- Fixed print
- Fixed URL
- Update to use test ID
- Fixed print

lexinspector

- Create submissions and submit tests
- Create submissions and submit tests

testsubmissionapi

- Fixed URL

Saturday, 16/03/2019

lexinspector

- Create new submissions, submit tests, list submissions for document
- Fix base URLs
- Serialization options

Monday, 18/03/2019

lx-test-store-api

- Use related fields
- Add tag field so that ID doesn't need to be used to refer to tests across applications. ID may change.
- Add tag field so that ID doesn't need to be used to refer to tests across applications. ID may change.

lexinspectorui

- Test submission with tests
- Test submission

Tuesday, 19/03/2019

lexinspector

- Add tag field to tests
- Accessor on tag

lexinspectorui

- Added tag field

languagealyzer

- Bind to test tag not ID

Wednesday, 20/03/2019

lexinspector

- Updated submission model
- Get individual submissions
- Get individual test submission

lexinspectorui

- Submit tests and view completion status

lx-test-store-api

- Include test information in serializer for submission test

Friday, 22/03/2019

lexinspector

- Changed how timeline data is generated
- Submit user ID with document
- Fix start/end of seed
- Corrected zero count

lexinspectorui

- Add non-functional power toggles
- Made some changes

lxingress

- Parse user ID

Saturday, 23/03/2019

languagealyzer

- Classify using random forest classifier
- Updated requirements
- Mark test as complete

lx-test-store-api

- Make RO

Monday, 25/03/2019

languagealyzer

- Don't use alpine linux

Tuesday, 26/03/2019

punctuality

- PoC

lexinspectorui

- Sync tag

Wednesday, 27/03/2019

lexinspector

- Raw content
- Sort test results so that the test with the last character index appears first
- Get test results

lx-test-store-api

- Submission results
- Test field on results
- Include in serializer

TextMapperAPI

- Text mapper
- Helm chart

Thursday, 28/03/2019

TextMapperAPI

- Edited comment
- Mapping working correctly
- Added another edge case to test

- Added comments to a test for the sake of sanity, renamed it to make its name more specific
- Added a comment, not sure how useful it is
- Delete commented code that was moved into a branch

punctuality

- Exchange
- Analyse bracket use
- Don't analyse brackets
- Pika
- Jenkins, helm, docker

lexinspectorui

- Hacky but working display of results
- minor formatting changes

Thursday, 04/04/2019

punctuality

- Use text sampling API

TextSamplingAPI

- Corrected length check
- Completed test suite
- Helm and Jenkins
- Swashbuckle
- Swagger
- Sentence snapping, startup configuration
- Basic sampling implementation, no sentence snapping
- PoC
- Ignore more files
- Updated image name
- Docker image, run tests

language-length-analysis

- PoC
- Deployment configuration

Friday, 05/04/2019

lxingress

- JSON debugging

TextSamplingAPI

- Don't allow sampling on a space character
- Test space characters
- Test space characters
- Fixed tests that started with spaces

Tuesday, 09/04/2019

LexspecUIAPI

- Project

Wednesday, 10/04/2019

LexspecUIAPI

- Corrected ingress
- Updated server
- Helm and Docker setup
- Healthz
- PoC
- Updated path

lexspecui

- init

Friday, 12/04/2019

lexspecui

- Login flow
- Vue Material
- View documents and results

LexspecUIAPI

- Updated healthcheck URL
- Latest document submission

Thursday, 18/04/2019

homoglyph-detection-model

- Max iterations
- Changed classifier
- Max iterations
- RFC
- Venv
- RFC
- PoC

homoglyphoscope

- PoC
- Requirements
- Updated deployment information
- Updated Dockerfile

Saturday, 20/04/2019

homoglyphoscope

- Content not whole sample
- Model
- Should end loop on exception

lexinspectorui

- Updated services
- Updated subheadings

punctuality

- Debugging print lines
- Die on exceptions

language-length-analysis

- Die on exception

Sunday, 21/04/2019

TextMapperAPI

- Compare two characters before and after to reduce chance of collisions

---

Tuesday, 23/04/2019

lexspecui

- Deployment setup
- Tweaks
- Removed unused component

lexinspector

- Fix off-by-one error. Could also be fixed by setting time to midnight

LexspecUIAPI

- Test store adapter

lexinspectorui

- Made UI slightly less bad
- Fix delete row bug
