Computer Methods and Programs in Biomedicine 211 (2021) 106398

Contents lists available at ScienceDirect

Computer Methods and Programs in Biomedicine

journal homepage: www.elsevier.com/locate/cmpb

Code-free cloud computing service to facilitate rapid biomedical digital signal processing and algorithm development

Michael R. Jennings[a,*], Colin Turner[a], Raymond R. Bond[a], Alan Kennedy[c], Ranul Thantilage[b], Mohand Tahar Kechadi[b], Nhien-An Le-Khac[b], James McLaughlin[a], Dewar D. Finlay[a]

[a] Ulster University, Belfast, UK
[b] University College Dublin, Dublin, Ireland
[c] Pulse AI Ltd., Belfast, UK

ARTICLE INFO

Article history:
Received 6 May 2021
Accepted 30 August 2021

2021 MSC:
68U01

Keywords:
Cloud computing
Code-free
Django framework

ABSTRACT

Background and Objective: Cloud computing has the ability to offload processing tasks to a remote computing resources. Presently, the majority of biomedical digital signal processing involves a ground-up approach by writing code in a variety of languages. This may reduce the time a researcher or health professional has to process data, while increasing the barrier to entry to those with little or no software development experience. In this study, we aim to provide a service capable of handling and processing biomedical data via a code-free interface. Furthermore, our solution should support multiple file formats and processing languages while saving user inputs for repeated use.

Methods: A web interface via the Python-based Django framework was developed with the potential to shorten the time taken to create an algorithm, encourage code reuse, and democratise digital signal processing tasks for non-technical users using a code-free user interface. A user can upload data, create an algorithm and download the result. Using discrete functions and multi-lingual scripts (e.g. MATLAB or Python), the user can manipulate data rapidly in a repeatable manner. Multiple data file formats are supported by a decision-based file handler and user authentication-based storage allocation method.

Results: The proposed system has been demonstrated as effective in handling multiple input data types in various programming languages, including Python and MATLAB. This, in turn, has the potential to reduce currently experienced bottlenecks in cross-platform development of bio-signal processing algorithms. The source code for this system has been made available to encourage reuse. A cloud service for digital signal processing has the ability to reduce the apparent complexity and abstract the need to understand the intricacies of signal processing.

Conclusion: We have introduced a web-based system capable of reducing the barrier to entry for inexperienced programmers. Furthermore, our system is reproducable and scalable for use in a variety of clinical or research fields.

1. Introduction

Server-side processing of biomedical signals is widely prevalent [1,2] to the extent where specific standards have been formulated to support aspects of this approach [3]. However, the sharing of code and processing techniques is limited by the both the skill of the user and the willingness of the developer to format their code comprehensibly. This could be indifference of a developer to format code legibly, or it could be aversion due to time constraints. For example, there are an abundance of digital signal processing (DSP) algorithms proliferating, but limited initiatives are making these accessible to others. Artificial Intelligence (AI) techniques garner significant interest in multidisciplinary research, however, the abstraction from the developer to the user causes a lack of uptake in many cases. Additionally, few developers are willing to share their models directly, forcing the user to recreate already existing software. Furthermore, whilst new software tools and techniques have made specialist domains like DSP more accessible the technical barrier to comprehending algorithms

* Corresponding author.
E-mail address: jennings-m5@ulster.ac.uk (M.R. Jennings).

https://doi.org/10.1016/j.cmpb.2021.106398
0169-2607/

based on advanced techniques remains high. Many junior developers find it more difficult to reuse or recreate another published algorithm due to a lack of information surrounding its use or a lack of clarity in the method. This may reduce the eagerness of others to begin research into a novel area or technique and further publication bias [4]. Additionally, it may discourage junior developers from furthering their understanding of the area since an extensive time investment is required.

Notwithstanding, non-developers could benefit most from the increased availability of well-documented code since they are often experts in the application of software, rather than it's development. One example is that from the medical domain. Specifically, a physician likely has little-to-no experience in software development, but they process large quantities of data on a daily bases. Much of this data is processed using experience and judgement learned through years of training, however, this training is not available to everyone. Additionally, many assignments undertaken by the clinician are, in fact, quite routine and could be automated, allowing them to focus time on other areas. A system that allows the clinician to offload the processing of data has the capability to reduce the decision-time overhead, potentially enabling more patient-centric care. Additionally, medical data science is often undertaken by non medical professionals. Removing the coding barrier may facilitate the discovery of new findings in medical science by increasing its availability.

Clinicians often have access to biomedical data such as the electrocardiogram (ECG) which require extensive filtering between capture and interpretation. Presently, cardiology professionals are required to manually review information and return a diagnosis or opinion. ECG traces may be seconds to hours in length for some ambulatory monitors, requiring considerable time for interpretation. Cloud-based approaches to interpretation of biomedical signals may be a solution to this. Previous research have tackled code-free server-side signal processing and data-visualisation [5–9], however, such approaches do not allow the user to develop their own algorithm or experiment with different functions. One solution may be an architecture that allows users to create algorithms by connecting multiple combinations of code or functions then expose it to data. Such a system allows the code to be reused; essential to overcome the 'replication crisis' in health informatics [10].

There are many different file formats that can be used containing patient data, although, they are generally not interchangeable [11]. Previous research have proposed middleware format conversion methods capable of adapting data to a universal format [12,13].

In the work reported in this article, we have developed a system capable of abstracting the user from the need to write code or possess a strong understanding of digital signal processing. This has been developed based on the notion of streamlining the algorithm development process and aiding code reuse by allowing the rapid configuration of new algorithms while supporting the repeatability of experiments. In this work, we have paid particular attention to the notion of abstracting the user from the various integration issues such as different programming languages, compatible functions, and data formats to allow them to focus on processing biomedical data. In addition to facilitating biomedical data processing functionality, the work proposed in this article also includes provision to facilitate storage of associated data. This data storage functionality has been incorporated to reduce the probability of data silos forming due to the distributed nature of sensitive information [14] by offering a central data store and development area for each user [15]. It is hoped that this may increase the willingness of healthcare providers to share information by removing incurred costs [16], potentially negating many of the data-sharing complications that result in inconsistent care [17].

Table 1
Sub-review of Code Sharing Prevalence in Literature.

Metric	Prevalence
Total Articles	25
Included Articles	13
Excluded Articles	12
Matching 'T' Criteria	3 (23%)
Matching 'F' Criteria	10 (77%)

Furthermore, our work will provide a more beginner-friendly system to those unfamiliar with DSP with the capability to further the democratisation of computational health informatics [18]. This platform aims to be language-agnostic by supporting multiple different programming languages e.g. MATLAB or Python. The suggested framework will offer a platform for peer review where code written by one author can be shared with another. This may reduce the impact by which applications created by non-experts with potentially unethical consequences have on the community [19] since the code can be compared against other 'gold standard' approaches.

2. Background

To investigate the context of source code sharing and reuse culture in academia, a sub-review was devised. The purpose of the sub-review was to quantify the proportion of authors who share their source code, program or provide an example application e.g. working website. Using Google Scholar, a search was devised using the terms "novel web framework". The top 25 results were used as the basis for the following rules. The inclusion criteria comprised of journal or conference papers with a clear indication a computerised method published in the past ten years (since 2011). Articles that did not fulfill the inclusion criteria were excluded ($n = 12$). Those matching the inclusion criteria ($n = 13$) were sorted into two categories: 'T', representing those linking their source code, website, or program in the article; 'F', representing those who did not meet the 'T' criteria. Those matching the 'T' ($n = 3$) and 'F' ($n = 10$) criteria comprised 23% and 77% of the included set respectively. These results are shown in Table 1.

Of the articles in our sub-review, only 23% shared their code, website or application. To facilitate the democratisation of DSP and code reuse, a higher number of authors must release their source code. This allows other researchers to not only evaluate the performance, but verify their results and collaborate on changes as part of the scientific method.

3. Methods

3.1. Framework

A framework abstracts common software functions and provides a template which can save development time [20]. Multiple frameworks were evaluated, with the Python-based Django framework being chosen. The Django framework was chosen as the basis of this study due to the apparent ease of use in creating web applications, primarily in the provision of an automatic graphical administrator (admin) interface to assist with database management. This allows an administrator to add and remove data without writing code. Additionally, it has a comprehensive documentation library and active user community to aid debugging. Furthermore, using a popular language such as Python may facilitate the reuse of this system by making it more accessible to those willing to recreate it. In the spirit of reusability, the source code of this project has been made available.

Django abstracts database creation to a number of potential SQL backends. The database schema is derived directly from classes within the source code, with items within the class informing columns and attributes within the database. This ensures consistency between the software and database schema and automated database migrations.

The Django framework is inherently open-source. This allows developers to view and edit aspects of the framework to suit their application [21]. In this study, we did not manipulate the Django request-handling middleware, however, it is an important feature when considering scaling a project or addressing architectural issues in future [22].

3.2. Database construction

The database structure is central to a cloud computing architecture. The purpose of the database, in this study, was to hold data files, code/executable files (scripts), user details and user inputs. Primarily, the database stored what data the user would like to process (File), what scripts to run (Script) and their order (Algorithm), and the result of each script (Execution). This allows the user to see upload data and process it repeatedly using either a novel or existing Algorithm they have created. In Django, database tables are referred to as models. Five database tables (models) were identified as core to this study:

Algorithm is the highest-level model in that it contains multiple other models within it. It is a user-created entry with a number of potential scripts to be executed in order. This model is linked to one user allowing them to document the order in which their uploaded data is processed. One field, *scripts*, is linked to a Script via a many to many relationship. This link is made through an intermediary table, Execution, which provides further details. One example of an Algorithm could be a disease classifier. An input file of patient data (XML) would be uploaded by the user. The data could be passed through two hypothetical scripts: first a MATLAB file 'data_sanitisation.m' and secondly a Python file 'knn_classifier.py' to return a spreadsheet or comma-separated variable (CSV) file with the result. In principle, however, an arbitrary number of such processes could employed within an Algorithm. This architecture allowed multiple combinations of the same Script to be called across various Algorithms without destroying or editing the original Script.

Execution is an object used to describe an instance of one file being processed by one script to produce an output file. Any data file being processed by a MATLAB or Python script will become part of an Execution. This model is hidden from the user. It contains one input file *data_input*, a script to execute the data *script*, and an output file *data_output*. The order in which each script was executed is stored in *order*. The purpose of this model was to separate each processing step of an Algorithm by handling the inputs and outputs of each Script individually. This allowed error handling, logging of output files and subsequently the removal of unused intermediary files.

Script is an executable file model. Its programming language (*language*), supported input file format (*data_input*) and output file format (*data_output*) are core fields. Only an admin can upload a Script to reduce the risk of malicious code injection. A *description* field is included to provide instructions for use and information as to how the Script works. The executable file is held in *uploaded_script* and is stored in a media file folder (/algorithms/). This allowed each executable file to be read-only by a user and so improved the application's security. Additionally, only an admin could edit An example of a Script might be a MATLAB low pass filter function that supports a single row CSV file and outputs the same type of file.

File is any file that can be attached to a user. For example, data files uploaded by the user or the result of an Execution. The user can provide a descriptive name (*name*) for the data and specify the data format (*format*). A example File may be a MATLAB data file ('.mat') of electrocardiograph (ECG) data with the *name* 'ecg_data.mat'

FileFormat contains metadata for a File instance. Primarily, this model is used to filter what Scripts are supported via a one to many relationship with the Script entries *data_input* and *data_output*. Also, FileFormat is used to store the MIME type for if the user downloads the file (*mime_type*). It is important to note the MIME type does not decide if a File is supported by a Script, that is handled by the administrator-controlled list of supported FileFormats for a given Script. The field *io* shows if the file is in input file, output file or both.

This schema allows users to create a library of data files in various formats, and build algorithms comprised of individual scripts to act upon them in a reproducible manner. This also makes the evolution and comparison of algorithms a more streamlined process.

Fig. 1 shows an entity relationship diagram (ERD) of each table (model).

3.3. Data upload

Data are handled in two discrete scenarios: upload and execution.

Only authenticated users can upload data. When accessing the file upload page, an empty instance of File is created. In the class-based approach of Django, this creates an empty row in the File table. The user is prompted to upload their file, give it a descriptive name and specify the input format. These were stored in a media directory (/user_data/) and assigned a filename corresponding to the username and a universally unique identifier (UUID) e.g. 'user_a535562....csv'. In this way data can be traced to a user by searching file structure or querying the database.

Data are passed to the controller using a POST request. If the form data and file upload were valid, the file would be saved within the system and the model instance of File updated with the user-provided information. The path to the uploaded file and meta information such as FileFormat could be accessed by a database query. Fig. 2 shows the process to upload user data.

3.4. Data processing

3.4.1. Algorithm creation

Creating an algorithm is handled in a similar way to uploading data. A blank HTML form was created with the following fields: Algorithm name, description, input data and scripts. The name and description are customisable to assist the user in keeping track of previous entries and to ensure a research team have a shared knowledge of the algorithm construction. The input data is derived from a selectable list of user data files. Only files from the current user are shown. The script form fields allow the user to select one or more Scripts in the desired execution order. Once submitted, a POST request is sent to the controller with the user-selected input data, executable scripts and a description from the algorithm creation form. In the model layer, an intermediary table was created to handle the ordering and metadata for each script. The Execution model was used for this. An instance of Execution was created for each chosen Script and the order assigned. This allows for additional Scripts to be added into the Algorithm construction at a later date. The input file chosen by the user is assigned to the first Execution. The input and output files for other Executions are set to null temporarily.

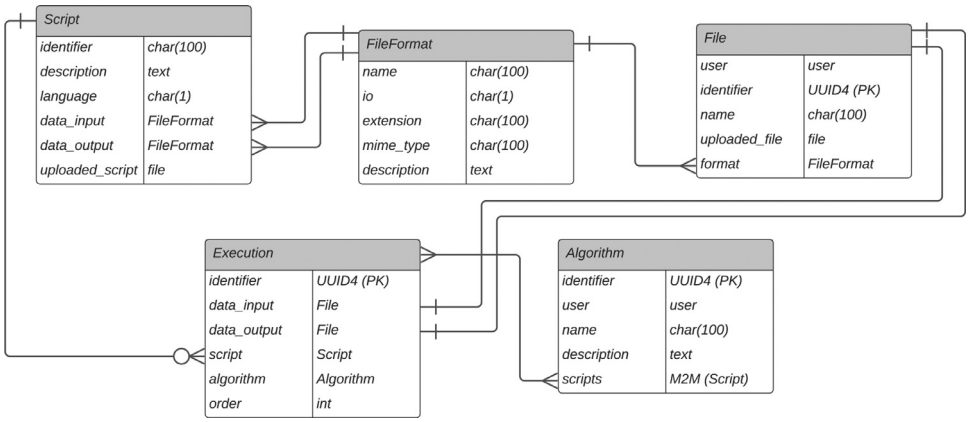

Fig. 1. Entity relationship diagram (ERD) of the algorithm development database. Each table represents a model in the Django framework. 'Algorithm' is the user-created entry consisting of a list of 'Scripts' which process 'Files' in the order set by 'Execution' providing they are a compatible 'FileFormat'.

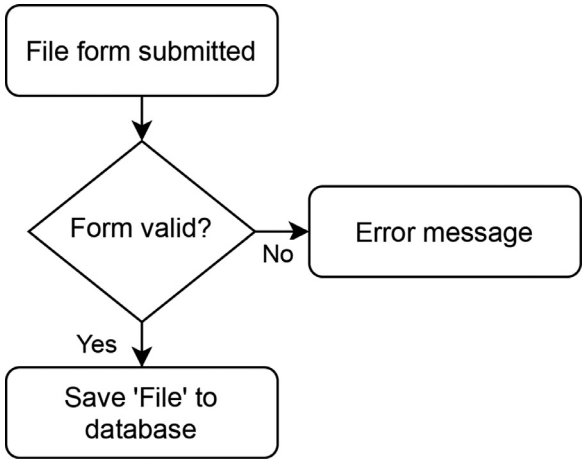

Fig. 2. Overview of the POST request checking following the user uploading a data file. *Note:* 'File' is a table (model) representing the data file and user metadata.

3.4.2. Execution

A complete instance of Execution contains an input data file, executable script and output data file. To complete the first instance of Execution for this Algorithm, the input data file was passed to the Script via a handler file. The handler file is different depending on the language of the Script. For example, a MATLAB file will have a 'handler.m' file and Python may have another file. The role of the handler file is to take the file path of the input data file, run a script at a given file path and return the file identification number. The output file was then assigned to the first Execution to complete it. The next Execution uses the output data file of the previous Execution as its input data file. It executes the script and returns an output file. If the last execution is reached, the output data file is returned to the user as a downloadable file and all previous intermediary files are cleared from the system to reduce storage overhead.

3.4.3. MATLAB engine for Python

The Django framework uses Python, so the native environment can be used to process scripts, however, the same is not true for licensed software such as MATLAB. The MATLAB Engine for Python is an application programming interface (API) for Python capable of accessing the MATLAB work space and executing scripts. This allows a licensed copy of MATLAB to be stored on the server with the current session shared with the Python virtual environment. Errors from MATLAB can be passed to Python via the API and raised to the user as a MatlabExecutionError. Fig. 3 shows a flowchart of how data is executed.

3.5. User interaction

A front-end was developed to facilitate testing. Separate webpages were created to demonstrate the following features: uploading and viewing data files, creating and viewing algorithms, and user registration.

When uploading a file, the user was presented with three input fields as an HTML form: "Name", "Uploaded file", and "Format". The user could view and manage their stored files via an HTML table including deleting unwanted data files. Each table row was an instance of File attributed to that user

To create an Algorithm, the user entered details into another HTML form. They could select a file from the file management area to process or enter it from the form directly. The following input fields were available: "Algorithm name", "Description", "Data input" and "Scripts". For the purposes of demonstration, up to four scripts were allowed, however an arbitrary number can be used within the model and administrator interface. Each successive script field denotes an instance of Script and allocated the order of each Execution instance. The data file selection is a filtered list of files for only that user. Once the user submits the Algorithm creation form, the data file will be processed in the manner previously described. The user will be requested to download the data output file. The user could also view and edit their created algorithms using the same method as data files. Fig. 4 shows the suggested user interaction with the system.

To show the user interaction in more detail, Fig. 5 provides a sequence diagram of user inputs followed by the backend response. Five objects have been described here. First, the web interface, is the frontend developed for testing purposes. This provides renders of forms such as file upload and algorithm creation forms. Second, the web application, is the backend model-view-controller architecture written using the Django framework. It handles user requests, renders and queries to the database. Third, the processing engine or API, describes the system which executes a given file by passing it to the selected script as an argument. The API will then return a result file and status message to indicate a successful execution. Finally, the database, is used to store the tables described in Section 3.2 and allows the user input to be preserved in case of an exception e.g. the processing API raises an error.

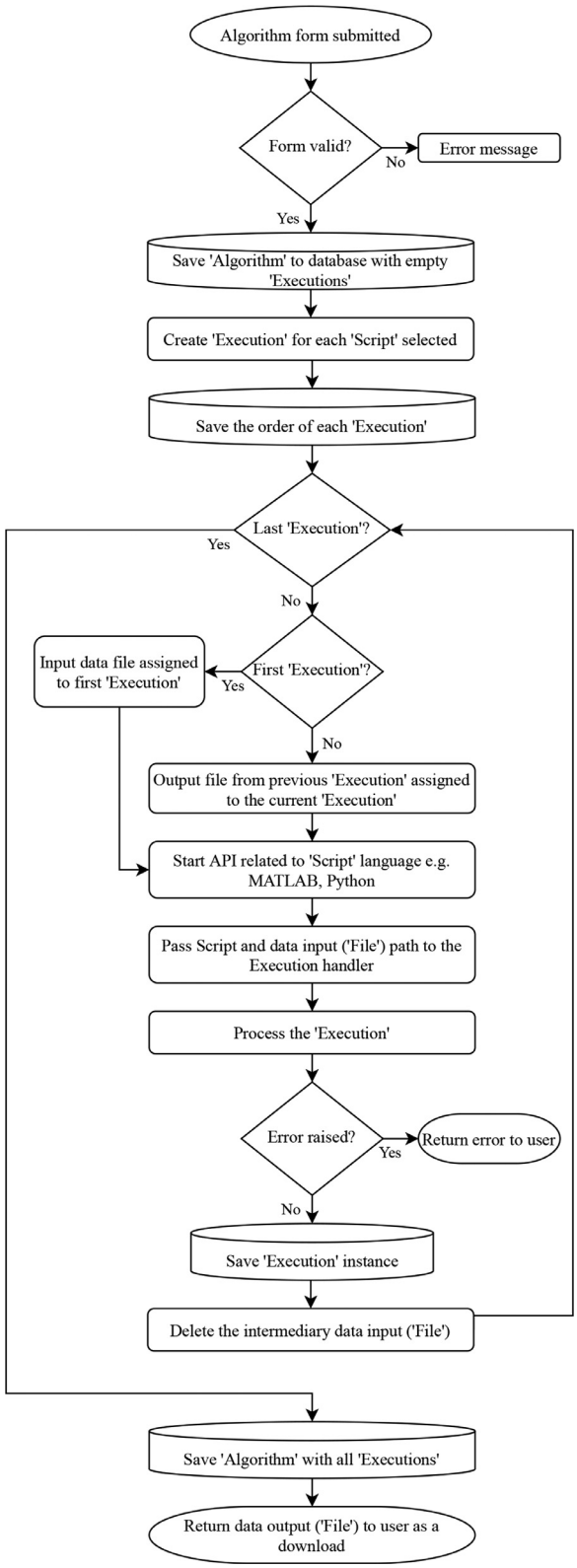

Fig. 3. Data flow following the submission of an 'Algorithm' form to process a user input file through multiple different scripts ('Execution') and return an output file.

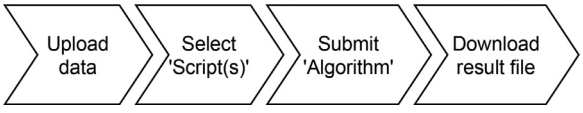

Fig. 4. User interaction steps required to process data.

4. Discussion

A web-based approach to algorithm development allows a user to trial many different parameters without writing code. For example, a user can combine different filters and compare which has the most favourable performance for their signal. Additionally, a user could compare classifiers written by different team members. This system is not limited to comparing scripts though, it could be utilised by medical professionals for statistical analysis or used by administrators to sort patient records without prior knowledge of programming.

Code reuse, particularly in open source software, has the capability to reduce development time and increase collaboration [23,24]. This system allows code to be stored in the form of discrete functions. Users can reuse a function multiple times and in different orders. In particular, a developer can see all previously created Algorithms or Scripts with metadata on the inputs, outputs and function of the program. If a Script has been previously created, there is no need for a developer to recreate it, thus, saving time during the development process. A function-based approach to Scripts abstracts the user from the coding aspect of algorithm design to promote a trial and error method where non-experts can experiment with their data. Additionally, a Script is linked to the user who uploaded it. In teams of developers or clinicians, this allows a potential user of the Script to contact the original author or team they are associated with. When compared with existing systems such as Apache Kafka, this system is complementary. Kafka implements an event-driven approach for data monitoring, however, this system employs a data-driven approach instigated by the user to allow experimentation. This may be beneficial in the design-phase of automation algorithms as a test-bench before using an event-driven architecture such as Kafka.

DSP software is often licensed. For a user to operate the software, they must purchase a licensed copy and activate it. Each user of requires a license which some institutions or companies may not be able to afford. Our system requires a single server license by running one copy of licensed software in the backend. Functions are called via the handler by using an API, negating the requirement for the user to have licensed software.

Many users may not have an in-depth understanding of file formats or DSP principles. There may be many errors when processing data. Licensed software APIs such as the MATLAB Engine for Python raise errors in the Django framework. This enables the development of an error handler to return exceptions to the user. This is a reactive error handling which happens after the error has occurred. A pro-active approach is to query the backend before submitting the Algorithm. Javascript in the form of an AJAX query was used to filter the supported Scripts available to the File-Format of the input data file. This would reduce the likelihood of import errors, however, it would not address run-time errors.

Using APIs and file handlers allows the use of multiple programming languages and data types. For example, the output of a MATLAB Execution may be a CSV file. This could be passed to a Python Execution and processed interchangeably. Providing the Script supports a particular data file, it can be executed without knowledge of the previous programming language which processed it. This may introduce an environment where DSP software development teams can write functions in multiple languages without the need for single-language specialists. When recruiting developers, this would increase the number of potential candidates for a role and encourage a deeper understanding of the DSP principles rather than a deep understanding of one programming language.

This application is user agnostic. A wider variety of individuals can interact without specialist knowledge of the software. For example, a medic could process patient records to show risk fac-

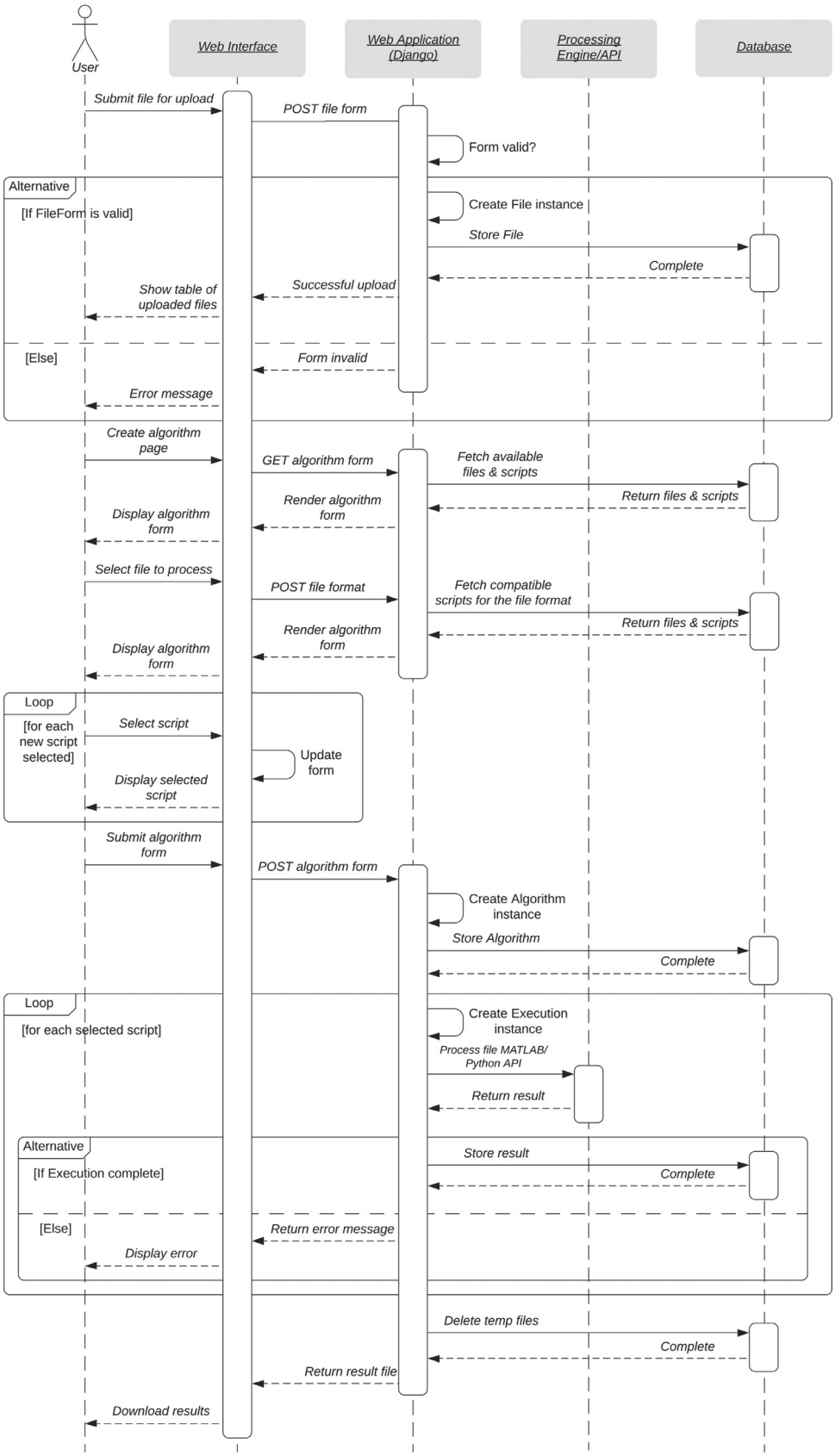

Fig. 5. Sequence diagram showing interactions between the user and the system when uploading a file, creating an algorithm and running the algorithm. User interface refers to the frontend browser-based platform. The web application is the backend model-view-controller logic. The processing engine is the API which uses a selected script to process a file e.g. MATLAB or Python. The database is the server used for storage and queries.

tors for a specific ailment. Likewise, an embedded systems engineer could design filter coefficients for a medical device to meet a regulatory requirement. Abstraction of these tasks from the user reduces the time invested in the task, allowing them to focus elsewhere. The principle of democratisation in software development is to allow any user to interact and access the core functionality. A user agnostic system by default achieves these principles.

Data files are uploaded by users to our system. Keeping data files in one system can reduce the probability of data silos forming, especially when the data storage is centralised. The user can provide descriptive information to describe their data file, potentially increasing the prospect of data reuse and collaboration.

In our system, only MATLAB and Python scripts have been tested. The architecture has the potential for many different programming languages and programs to execute arbitrary scripts in various languages such as R, C++ and Java, but with administrative safety measures. Additionally, this system could be used to output typographic information by employing TeX-based compilers. One utilisation instance of typographic processing could be the cleaning of patient record files to output a formatted table or document.

Executable scripts can only be uploaded by admins or superusers. This reduces the risk of malicious code injection by only allowing users to upload data files. Additionally, it ensures that only approved executable files are included in the Script database. In a software development environment, this would be post code-review and could reduce the number of errors experienced by clients.

This architecture is inherently scalable. An object-based approach to processing scripts allows multiple instances of 'Executions', each with their own engine. For example, multiple MATLAB engines or Python environments can be created, each with separate memory. Since the memory space is not shared by these 'Executions', they can be containerised using platforms such as Docker and Kubernetes. These systems allow scaling to occur automatically while processing data in parallel to the main web application thread.

The file management system links all files with a user. This enables efficient clearing of old or disused files by the user or by an administrator. Additionally, data associated with the user can easily be collated or removed to comply with right-to-erasure requests such as GDPR or similar 'right to be forgotten' requests.

Assessing the security and vulnerabilities of a code is an important factor due to the increase in global cyber security threats. The code has been subjected to vulnerability scanning using Bandit, a popular tool used to detect known common issues in Python code. According to the scan the security risk of the code is considered 'medium' due to the MATLAB engine requiring the use of command-line tools.

5. Limitations

Remote code execution (RCE) presents a considerable security concern in web-based applications. Malicious code can be injected to such a system and potentially lead to compromise of the system [25]. The Django framework provides features to improve security such as cross site scripting (XSS) and SQL injection protection [26], however specific protection measures would be required for the deployment environment.

Regular expressions (regex) and user input sanitisation was limited during this study in the interest of time. For this application to be deployed and secured, care would be taken to reduce the likelihood of string-based injection attacks by parsing user files for executable scripts [27].

At present, handler functions are used to execute scripts. The handler function is passed an absolute file path to the data and script files. This requires a file to be present for each execution instance. For less complex scripts, much of the processing time would be allocated to reading and writing data files. It would be more efficient to use the Django framework to handle the files as imported variables, however this was beyond the scope of this study.

6. Future work

Many DSP functions collate multiple data files during execution. For example, combining ECG waveforms and contextual patient metadata to produce a patient-specific diagnosis. This would require multiple data files for each instance of Execution, necessitating a database architecture change. Following this, a user could select multiple data files of different file formats for one script. This may allow more context to be given to classifiers.

To reduce the risk of damage due to malicious code injection, a sandboxing method could be employed. Sandboxing can isolate server instances to that user or group of users to assist with malware detection [28]. A compromised sandbox instance will damage the virtual machine (VM) it is incased within, however, it is less likely to affect other sandboxes due to their distributed nature.

Automation of file upload and execution could be handled by the development of an API. More specifically, medical devices and embedded systems could use this architecture to offload processing requirements to the cloud. This can reduce the size and cost of hardware required while allowing algorithm and software changes to be handled remotely, reducing the need for product recall and firmware updates. Medical devices such as Holter monitors could upload ECG data and have near to real-time decisions using this approach. Additionally, all patient data would be accumulated in one location, lessening the data silo effect. Cloud storage systems could be linked such as Microsoft OneDrive or Google Drive to further improve the centralisation of data. However, this would be limited by regulations on patient data sharing with third parties.

Error handling in this study is limited, however a separate error handling user interface could allow users to debug data files and code simultaneously. Furthermore, the handler could include a conditional-based flow during the Execution phase whereby a certain output may trigger a response. For example, if a single row of a patient record is missing it could be estimated by another script instead of raising an error.

7. Conclusions

In this study, we have presented an adaptive cloud computing architecture capable of processing arbitrary input files through ordered executable scripts using multiple processing languages in a repeatable manner. Using the Django framework, a database was introduced to handle and store files as they are processed. This work has the capability to assist algorithm research teams during development by reducing the time taken to incorporate previously developed code. Additionally, this study provided an insight into the potential for automation to process IoT device data, particularly long-term patient monitoring systems.

Funding

This project is part of the Eastern Corridor Medical Engineering centre (ECME). It is supported by the European Union's INTERREG VA Programme, managed by the Special EU Programmes Body (SEUPB). Funding number: IVA5034.

Declaration of Competing Interest

Dr Alan Kennedy is the founder of Pulse AI Ltd., a company specialising in cloud processing of ECG. We can confirm that Dr

Kennedy receives no benefit from this project and was involved as an advisory role only. No other authors have conflicts of interest to disclose.

Supplementary material

Supplementary material associated with this article can be found, in the online version, at doi:10.1016/j.cmpb.2021.106398.

References

[1] H. Xia, I. Asif, X. Zhao, Cloud-ECG for real time ECG monitoring and analysis, Comput. Methods Programs Biomed. 110 (3) (2013) 253–259, doi:10.1016/j.cmpb.2012.11.008.

[2] S. Pandey, W. Voorsluys, S. Niu, A. Khandoker, R. Buyya, An autonomic cloud environment for hosting ECG data analysis services, in: Future Generation Computer Systems, vol. 28, North-Holland, 2012, pp. 147–154, doi:10.1016/j.future.2011.04.022.

[3] ISO, ISO/IEEE International Standard for Health informatics – Point-of-care medical device communication – Part 20702: medical devices communication profile for web services, in: ISO/IEEE 11073-20702:2018(E), 2018, pp. 1–52, doi:10.1109/IEEESTD.2018.8472336.

[4] A. Cockburn, P. Dragicevic, L. Besançon, C. Gutwin, Threats of a replication crisis in empirical computer science, 2020, 10.1145/3360311

[5] M.A. Kumar, A. Srinivasan, N. Bussa, HTML5 powered web application for telecardiology: a case study using ECGs, in: IEEE EMBS Special Topic Conference on Point-of-Care (POC) Healthcare Technologies: Synergy Towards Better Global Healthcare, PHT 2013, 2013, pp. 156–159, doi:10.1109/PHT.2013.6461308.

[6] X. Wang, Q. Gui, B. Liu, Z. Jin, Y. Chen, Enabling smart personalized healthcare: a hybrid mobile-cloud approach for ECG telemonitoring, IEEE J. Biomed. Health Inf. 18 (3) (2014) 739–745, doi:10.1109/JBHI.2013.2286157.

[7] L. Jin, J. Dong, Intelligent health vessel ABC-DE: an electrocardiogram cloud computing service, IEEE Trans. Cloud Comput. (2018), doi:10.1109/TCC.2018.2825390.

[8] A. Joshi, D. Scheinost, H. Okuda, D. Belhachemi, I. Murphy, L.H. Staib, X. Papademetris, Unified framework for development, deployment and robust testing of neuroimaging algorithms, Neuroinformatics 9 (1) (2011) 69–84, doi:10.1007/s12021-010-9092-8.

[9] E.S. Martin, D.D. Finlay, C.D. Nugent, R.R. Bond, C.J. Breen, An interactive tool for the evaluation of ECG visualisation formats, in: Computing in Cardiology, IEEE, Zaragoza, Spain, 2013, pp. 779–782.

[10] E. Coiera, E. Ammenwerth, A. Georgiou, F. Magrabi, Does health informatics have a replication crisis? J. Am. Med. Inf.Assoc. 25 (8) (2018) 963–968, doi:10.1093/jamia/ocy028.

[11] R.R. Bond, D.D. Finlay, C.D. Nugent, G. Moore, A review of ECG storage formats, Int. J. Med. Inf. 80 (10) (2011) 681–697, doi:10.1016/j.ijmedinf.2011.06.008.

[12] X. Li, V. Vojisavljevic, Q. Fang, An XML based middleware for ECG format conversion, in: Proceedings of the 31st Annual International Conference of the IEEE Engineering in Medicine and Biology Society: Engineering the Future of Biomedicine, EMBC 2009, IEEE Computer Society, 2009, pp. 1691–1694, doi:10.1109/IEMBS.2009.5333907.

[13] R.R. Bond, D.D. Finlay, C.D. Nugent, G. Moore, XML-BSPM: an XML format for storing body surface potential map recordings, BMC Med. Inf. Decis. Making 10 (1) (2010) 28, doi:10.1186/1472-6947-10-28.

[14] R. Ranchal, P. Bastide, X. Wang, A. Gkoulalas-Divanis, M. Mehra, S. Bakthavachalam, H. Lei, A. Mohindra, Disrupting healthcare silos: addressing data volume, velocity and variety with a cloud-native healthcare data ingestion service, IEEE J. Biomed. Health Inf. 24 (11) (2020) 3182–3188, doi:10.1109/JBHI.2020.3001518.

[15] R. Reda, F. Piccinini, A. Carbonaro, Towards consistent data representation in the IoT healthcare landscape, in: ACM International Conference Proceeding Series, vol. 2018-April, Association for Computing Machinery, New York, NY, USA, 2018, pp. 5–10, doi:10.1145/3194658.3194668.

[16] A.R. Miller, C. Tucker, Health information exchange, system size and information silos, J. Health Econ. 33 (1) (2014) 28–42, doi:10.1016/j.jhealeco.2013.10.004.

[17] R. Reda, F. Piccinini, A. Carbonaro, Semantic modelling of smart healthcare data, in: Advances in Intelligent Systems and Computing, vol. 869, Springer Verlag, 2018, pp. 399–411, doi:10.1007/978-3-030-01057-7_32.

[18] Computational health informatics in the big data age: a survey, 201610.1145/2932707

[19] R. Bond, A. Koene, A. Dix, J. Boger, M.D. Mulvenna, M. Galushka, B.W. Bradley, F. Browne, H. Wang, A. Wong, Democratisation of usable machine learning in computer vision, 2019,

[20] O. David, J.C. Ascough, W. Lloyd, T.R. Green, K.W. Rojas, G.H. Leavesley, L.R. Ahuja, A software engineering perspective on environmental modeling framework design: The Object Modeling System, Environ. Model. Softw. 39 (2013) 201–213, doi:10.1016/j.envsoft.2012.03.006.

[21] S. WEBER, The Success of Open Source, Harvard University Press, 2009.

[22] D.M. Le, D. Link, A. Shahbazian, N. Medvidovic, An empirical study of architectural decay in open-source software, in: Proceedings - 2018 IEEE 15th International Conference on Software Architecture, ICSA 2018, Institute of Electrical and Electronics Engineers Inc., 2018, pp. 176–185, doi:10.1109/ICSA.2018.00027.

[23] S. Haefliger, G. Von Krogh, S. Spaeth, Code reuse in open source software, Manage. Sci. 54 (1) (2008) 180–193, doi:10.1287/mnsc.1070.0748.

[24] A. Mockus, Large-scale code reuse in open source software, in: First International Workshop on Emerging Trends in FLOSS Research and Development, FLOSS'07, IEEE Computer Society, 2007, pp. 7–11, doi:10.1109/FLOSS.2007.10.

[25] M.M. Hassan, U. Mustain, S. Khatun, M.S.A. Karim, N. Nishat, M. Rahman, Quantitative assessment of remote code execution vulnerability in web apps, in: Proceedings of the 5th International Conference on Electrical, Control & Computer Engineering, vol. 632, Springer, Kuantan, Pahang, Malaysia, 2020, pp. 633–642, doi:10.1007/978-981-15-2317-5_53.

[26] Django Software Foundation, Security in Django, 2017,

[27] Y. Zheng, X. Zhang, Path sensitive static analysis of web applications for remote code execution vulnerability detection, in: Proceedings - International Conference on Software Engineering, 2013, pp. 652–661, doi:10.1109/ICSE.2013.6606611.

[28] M. Vasilescu, L. Gheorghe, N. Tapus, Practical malware analysis based on sandboxing, in: Proceedings - RoEduNet IEEE International Conference, IEEE Computer Society, 2014, pp. 1–6, doi:10.1109/RoEduNet-RENAM.2014.6955304.