A close-up of a computer science project

AI-generated content may be incorrect.

Bioplatform: a full-stack multiple sequence aligner and phylogenetic tree tool

Bryce Johnson

Professor Rong Jin

Table of Contents

[Abstract 2](#_Toc198071998)

[Key Terms 2](#_Toc198071999)

[Introduction & Background 2](#_Toc198072000)

[Problem Definition 2](#_Toc198072001)

[Objectives 4](#_Toc198072002)

[Development Environment 5](#_Toc198072003)

[Operational Environment 6](#_Toc198072004)

[Requirements Descriptions 6](#_Toc198072005)

[Frontend Requirements 6](#_Toc198072006)

[Backend Requirements 6](#_Toc198072007)

[Deployment Requirements 7](#_Toc198072008)

[Design Description 7](#_Toc198072009)

[Figure 1. 7](#_Toc198072010)

[Figure 2. 9](#_Toc198072011)

[Figure 3. 10](#_Toc198072012)

[Implementation 10](#_Toc198072013)

[Figure 4. 11](#_Toc198072014)

[Figure 5. 14](#_Toc198072015)

[Testing and Integration 16](#_Toc198072016)

[Difficulties and Limitations 17](#_Toc198072017)

[Installation Instructions 19](#_Toc198072018)

[Operating Instructions 19](#_Toc198072019)

[Recommendations for Enhancement 20](#_Toc198072020)

[Bibliography 23](#_Toc198072021)

# Abstract

In the field of bioinformatics, the multiple sequence alignment is a staple tool for the analysis and comparison of genetic sequences. The result of a MSA is often utilized to create phylogenetic trees, but many existing multiple sequence aligners and phylogenetic tree generation tools exist as separate tools or require powerful hardware. Additionally, current solutions fail to leverage the existing artificial intelligence and associated deep learning technologies of today. This paper proposes bioplatform, a full stack application is proposed for the purpose of consolidating the analysis of nucleotide sequences into a single hosted user experience. The codebase is available at https://github.com/Baltharaaz/bioplat

# Key Terms

Multiple Sequence Alignment – an analysis method of bioinformatics that attempts to align two or more sequences according to their similarity. Multiple sequence alignments are conducted at the discretion of specialized bioinformatics analysists who cultivate specific datasets to determine the relationships between different nucleotide and amino acid sequences

Phylogenetics – a study of the evolutionary relationships between different biological units, whether that be genes, proteins, or whole organisms. Often done through a phylogenetic tree, which is a visualization of the relationship between various entities. In bioinformatics, these often represent the diverging points of a common ancestor.

Homology – in regard to bioinformatics, the similarity between different protein amino acid or nucleotide sequences. When discussing homology in this capacity, the sequences are considered if they are paralogs, sequences that have diverged from each other as a result of a gene duplication event, or orthologs, sequences that have originated from a common ancestor and diverged due to a speciation event.

# Introduction & Background

## Problem Definition

The multiple sequence alignment (MSA) is an analysis tool designed to aggregate and align various protein or nucleotide sequences of interest. It serves as an essential cornerstone for bioinformatics and subsequent phylogenetics. In tandem with the discretion of the user, the results of an MSA provide an opportunity to investigate homology of orthologs or paralogs through visualizations like phylogenetic trees. In tandem with the phylogenetic visualization, multiple sequence alignments serve as powerful analytical tools to discover relational information about any sequence data batch. To execute an MSA, statisticians typically utilize one of several alignment methods: dynamic programming, progressive alignment construction, iterative methods, consensus methods, hidden Markov models, phylogeny-aware methods, motif finding, and non-coding multiple sequence alignments. Each methodology has seen the creation and publication of associated algorithms that are available for utilization. MSA methods, however, are often subject to the success of the earliest aligned sequences in the batch; if any errors occur at this step for the analysis the user is attempting, they will propagate forwards.

Many of the web-hosted implementations of these algorithms exist separate from one another, necessitating the user submit to each algorithm they wish to utilize. They often only provide their specific algorithmic implementation with no consideration of additional tools, such as phylogenetic tree creation. Instances where multiple different MSA algorithms are available are often restricted to specific bioinformatic tools and programs, which do bundle a plethora of analytical tools for the end user. Yet, these require the user to possess sufficient hardware for the execution of these tasks on a device, which can be problematic when MSA datasets grow in size. It is not unheard of for MSA tasks to take hours or even days to execute given a large number of sequences and their varying levels of complexity.

With the advancement of artificial intelligence methods, some would argue that deep learning techniques are suitable for the analysis of biological data, due to the large and diverse datasets as well as the need for feature identification that may otherwise require domain knowledge and manual feature extraction (Yousef & Allmer, 2023). Some efforts have been made into incorporating deep learning tools into bioinformatic problem solving, with a particular focus on the protein structure prediction space (Abramson et al., 2024; Sapoval et al., 2022). Though there have been successes in incorporating those deep learning techniques, they are still subject to the black-box nature of the deep learning subspace. Results, though they offer useful insights into the datasets, are subject to scrutiny as a result of their inexplicability. Additionally, the focus of deep learning techniques on the medical and protein visualization spaces has left DNA nucleotide and amino acid sequence analysis somewhat lacking in comparison. The medical and protein focuses of the MSA space demonstrate the increasing application and acceptance of deep learning techniques in the biological problem space, and subsequent lack of nucleotide sequence and multiple sequence alignment deep learning solutions demonstrate a gap in bioinformatics domain knowledge. Similarly, phylogenetic deep learning solutions remain limited, with some considered phylogenetic methods still targeting the medical domain for the purpose of tracing back disease proliferation via phylogenetic hierarchies (Sun et al., 2024).

The deep learning MSA bioinformatics space is not entirely unexplored; some methods for deep learning MSA algorithms have been proposed recently (within the last five years). These include the BetaAlign method, a natural language processing (NLP) that utilizes transformers to take input sequences, concatenating them into a single sentence representation for processing, and then providing a “translated” sentence as the aligned sequence (Dotan et al., 2025). By constructing an NLP model on unaligned sequences and their associated “aligned” translation, Dotan et al. leverage the strength of the NLP feature detection within the nucleotide MSA domain. However, the model is fairly new, with the publication dated to the beginning of this year; the publication lacks forward citations and real-world applications. Another publication of note is the proposal of DPAMSA by Liu et al., which utilizes deep reinforcement learning for alignment accuracy enhancement (2023). While promising, this method is similarly new with only three forward citations. Only one of these forward citations details another MSA tool, and said tool is recently proposed as a preprint article. Deep multiple sequence alignment, therefore, remains a largely unexplored and untested domain, making incorporation into existing production solutions questionable. To consider the applicability of these solutions, more research will be required.

Multiple sequence alignments and phylogenetics, therefore, lack a merged, widely available solution that incorporates deep learning. This solution would incorporate both analysis tools into a single application that hosts and executes MSA jobs and their results. Additionally, this application’s available MSA algorithms should consider new deep learning alternatives to existing algorithms, due to the scale of data and feature recognition involved in MSA tasks. The application would include such an algorithm alongside commonly accepted MSA strategies. If incorporated, these deep learning strategies should be investigated to potentially improve the explicability and transparency of their execution.

## Objectives

The main objective of this project, therefore, is to provide a unified full-stack application that can accept sequences from an end user, complete a multiple sequence alignment, generate a corresponding phylogenetic tree, and save all of these for the end user. A user must be able to register for an account to begin submitting MSA jobs into the platform. Once they log into the platform, they should receive their existing completed jobs – the submitted unaligned sequences, the resulting aligned sequences, and the corresponding phylogenetic tree – on their user page. Additionally, they should be capable of deleting these job entries upon no longer needing them within the database. The end application must be capable of these requirements.

Additionally, it is a strong desire for the algorithms running the multiple sequence alignment to be powered by deep learning techniques to leverage the feature identification and extraction properties, as well as reinforce the correctness of alignments via deep learning’s self-improvement. Analysis of recently proposed deep learning techniques in the MSA space will be conducted to determine the applicability of these tools to the full-stack application. If incorporated, these deep learning solutions should also be considered for methods that may expose their black-box behaviors such as LIME (Riberio et al, 2016). These methods impart explicability that can then be incorporated into MSA job results. To accomplish this, BetaAlign will be investigated first to determine the applicability of the deep learning solution to the application’s MSA datasets; if it proves insufficient, the DPAMSA method will be subsequently considered. If deep learning methods fail to be incorporated into the runtime of the full-stack application, future consideration and building of a customized model should be considered to expand the available MSA options of the bioplatform.

The application is to provide an easy to navigate and readable user interface for interactivity. Creation of accounts, execution of logging in, and submission of jobs from the front-end interface should be simple and straightforward for end users. Similarly, the back-end API should be constructed in a manner to allow for easy addition of functionality and endpoints if necessary for current or future functionality. The entire application shall be constructed using commonly employed frameworks to allow for easy future iteration, whether from myself or another party. No code generation via AI assistants is to be utilized to promote familiarization with the currently widely accepted programming frameworks, toolsets, and modules.

## Development Environment

For the development of the project, the hardware environment utilized was a powerful home computer in an attempt to demonstrate the ability to run the entire application on accessible, if somewhat pricy hardware. The processor utilized is a 12th generation Intel i9-12900H processor with 14 cores running at base 2.50 GHz speed, with a boosted speed of 5.00 GHz. The device also possesses a strong GPU unit: an NVIDIA GeForce RTX 3080 Ti with 16 GB of dedicated visual RAM. The device also possesses 32 GB of 4800 MHz dedicated RAM. The device possesses an internal solid state drive with 1 TB of storage, as well as two external solid state drives with 2 TBs of storage each. This combination of hardware ensures that any potential deep learning model training is nearly as fast as possible on commercially available hardware.

For the software environment, the front-end application is built in the JetBrains WebStorm JavaScript IDE. This IDE possesses user made packages to enhance functionality, strong code suggestion and completion mirroring industry standard practices, and excellent warning and error notification to handle bugs or other issues before compilation and runtime. Additionally, it was chosen due to being freely accessible for students of California State University – Fullerton. JavaScript itself was chosen due to easy HTML manipulation to serve a website interface available to end users, as well as simple stylization through accompanying CSS.

Similarly, the back-end application component is built within the JetBrains PyCharm Python IDE. The IDE is chosen for similar reasons as above: easy package integration, code completion, error detections, and debugging systems. Python was chosen for the programming environment due to the existence of strong bioinformatics and deep learning packages, as well as the existence of strong REST API building frameworks. Familiarity with the Python programming language was also a strong motivation behind the selection.

Additionally, GitHub is utilized for code commit tracking, continuous integration and continuous deployment (CI/CD) pipeline testing, and branch control. It provides a strong platform for the present and future development of the program, as well as easy incorporation of open-source contributions. For scheduling, issue tracking, and agile development planning, Atlassian’s Jira solution is leveraged. Jira allows users to create tasks, assign them to dedicated sprint development periods, and ensure deadlines are met during production.

## Operational Environment

The operational hardware environment of the application is identical to the development environment of the application. This is a result of unfinished deployment to a hosted source; ideally, the application would exist on a cloud computing solution such as Microsoft Azure or Amazon Web Services. This was not accomplished due to financial hurdles and time constraints. The entire full-stack application is capable of running on the same device. However, an end user would be capable of accessing the application from anywhere, as it is a web-hosted source that can be associated with a proper URL and accessed via web browser for utilization anywhere with a network connection.

The most notable operational environment restriction is that whatever hardware hosts the back-end API application must also possess the corresponding MSA software and proper PATH environment variable configuration to execute each MSA algorithm, as the back-end is reliant on Python subprocesses to execute and handle MSA returns. Regarding this, it is probably most suitable for the back-end application to run on a custom-configured hardware unit that has the packages and environment adequately reassessed upon the inclusion of any additional MSA methods. This allows for easy expansion of the platform for new MSA functionalities.

# Requirements Descriptions

## Frontend Requirements

The platform shall provide an easily usable web interface with navigational ability available on every route in the front-end space. The interface will allow users to supply their own datasets for processing via their user account page. The interface will allow users to view their run jobs and results through saved records. If deep learning explicability methods are later incorporated, these results shall be accompanied by their relevant explanations. The interface will display additional data derived from run jobs through other bioinformatics solutions such as protein structure prediction, phylogenetic tree generation, and homology inference upon future addition. The interface should allow users to supply feedback on the functionality of the platform. The interface should allow and authenticate user login to allow viewing of user submitted jobs and results.

## Backend Requirements

The platform should, if necessary for the runtime of an MSA algorithm, acquire and aggregate data from established bioinformatics data sources to ensure quality and veracity of the data. The platform shall incorporate user-submitted data sets upon user request into the growing database of records. These user jobs shall be associated with each specific user to track user accounts and submissions. The platform shall utilize the most current methods for data handling regarding security and performance. The platform shall be capable of incorporating additional MSA algorithms in the future with minimal adjustment. The platform shall implement data handling through an API with administrative capabilities to view, delete, and create relations. The platform shall, if possible, incorporate a deep learning MSA algorithm as an option for end user jobs.

## Deployment Requirements

The platform shall integrate its React frontend layer with the Django framework to provide a feature-complete, seamless integration between the frontend and backend layers. The platform shall integrate with cloud hosted data solutions for asset hosting. The platform shall access and integrate data source APIs to fetch new data and enhance the capabilities of any deep learning multiple sequence alignment algorithms, if applicable.

# Design Description

## Figure 1.

A screenshot of a computer

AI-generated content may be incorrect.

The about page of the bioplatform application front-end. Contains a simple explanation of the platform alongside a navigation bar present on every page of the application.

The application is designed as a full-stack program, segmented into front-end and back-end components. For the front-end components, the application utilizes Vite, React.js, React Router, and TailwindCSS frameworks to construct the web interface. Vite is a local development server solution that allows plugins to be included within the final project production. React.js is an industry standard JavaScript framework that allows simple access and manipulation of page elements through the construction of functional components. The newest versions of React Router follow similar framework-style implementation to provide developers with quick construction of web application routes, including nested routes and layout construction. Additionally, TailwindCSS is utilized as a HTML stylization package. This module allows for easy on demand stylization of HTML elemenets using their className attribute to avoid CSS file bloat and isolate usages of specific stylization elements. TailwindCSS still allows for the utilization of associated CSS files with JavaScript component files, streamlining development of components while still leveraging the strengths of utilizing a dedicated CSS for applicable instances. TypeScript is utilized to reinforce type safety and catch potential errors prior to compilation and runtime. The front-end is designed to be a web application with a landing dashboard. The application is configured such that all pages are nested into a navigation bar “layout” to ensure the persistence of the navigation bar on each page of the application. The front-end is split into routes that correspond to the different potential URL locations: home, account, login, logout, and about. React Router allows routes to be declared in a specified route file to easily incorporate them into the front-end domain URL; future pages and routes can be added with ease. Subcomponents are defined based upon the reusability of code; these include the navigation bar, login/registration form, and the MSA submission form. This allows for easy embedding of these functionalities into routes. The login and register pages use the form component alongside passed props to reconfigure the API request logic in the form component and redirect accordingly. A protected route component exists to shield routes not accessible to unauthorized users. These routes need only wrapped in the protected route implementation to redirect nonauthorized users to the home dashboard. API requests are handled through the Axios library, which is configured to append the JWT token to every API request made from the frontend application.

For the database environment, PostgreSQL was chosen. PostgreSQL is an excellent extension of the typical SQL database architecture to handle more flexible data types, such as unbounded text strings. It still maintains the excellent performance and domain specialization SQL is known for in regards to handling structured data, which the program’s sequence alignment jobs qualify as. The proposal for the PostgreSQL relations includes a User relation that contains account information as well as a Job relation that contains the user’s submitted sequences and an associated name or title alongside the processed alignment and path to an image file. The application implements image file storage in this way due to the problematic nature of containing image files within database solutions such as PostgreSQL. PostgreSQL data may be accessed via the command line and modified with general SQL queries.

For the back-end API, Django alongside Djangorestframework and Djangorestframewort-simplejwt are utilized to construct a robust RESTful API engine that hooks into the PostgreSQL database server. Due to the implementation of Django, administrators need not directly access and query the PostgreSQL database to construct the necessary relations for the application. Instead, the programmatic implementation is to construct models in Python code that correspond to relations/tables within the database. Once models are constructed, the methods contained in the manage file are utilized via the command line to create and propagate the migrations of the models to the database server. Models are accompanied by a corresponding serializer that defines the method by which the database data can be converted to and from JSON formatting for data handling. These serializers are utilized by declared views that the user interfaces with via API urls. These views have generic baselines that implement API methods such as GET and POST. However, these generics may be overloaded to handle specific functionality. It is this overloading feature that allows the integration of ETE3 and MSA technologies. The views are then associated with programmer defined API urls, which use the view configuration to limit access to the API endpoint based on the authentication specification and accept the correctly configured requests from the front-end. Additionally, to host the entire platform on a single device for development, the back-end utilizes cross-origin resource sharing (CORS) headers middleware to ensure that the front-end and back-end can communicate with each other seamlessly. However, this should not be necessary in a full-scale deployment of the application.

## Figure 2.

A screenshot of a computer

AI-generated content may be incorrect.

Djangorestframework’s exposed API web. Depicted is the JWT authentication token endpoint with valid credentials passed to generate an access and refresh token.

Effectively, the back-end stack logic is constructed by creating a Job model that is associated with the User model via foreign key. This job model consists of a hidden ID primary key, a user supplied title, the user submitted unaligned sequences, the aligned sequence, and the path to the generated phylogenetic tree image file. An associated JobSerializer is created to define valid JSON representations of a Job model; this serializer declares that the author, aligned sequences, and phylogenetic tree path are read only to ensure the user does not submit them to the program. With the serializer defined, three views are created for the Job: one for creation and user job viewing, individual record deletion, and individual record viewing. To elaborate, these are responsible for getting all a user’s jobs on GET request and creating a job if a valid POST request is received, deleting a user’s Job given a specific Job ID, and getting a specific Job given its ID, respectively. The GET views are overloaded to filter out all Job results that do not correspond to the requesting User, which is supported via the JWT token embedded in the request header as it is associated with the current account instance for a given user. The POST method of the Job creation view is overloaded to extract the name and unaligned sequence from the serialized dataset. These tools are utilized alongside the ETE3 and subprocess modules to dispatch the data to an MSA job using temporary files. These views are then associated with API urls, upon which they can be called by the front-end or any sufficiently authenticated user with the ability to configure the necessary API request headers and data.

## Figure 3.

A screenshot of a computer code

AI-generated content may be incorrect.

An example of a phylogenetic tree generated from a multiple sequence alignment using ETE3 package

# Implementation

For the implementation of the front-end packages, the front-end environment exists in a single folder comprised of configuration files and subsections. All modules are contained with a node-modules folder for the current node version of 23.11.0. At the root of this front-end folder exists a docker configuration file to facilitate the deployment of the web application to a containerized solution when the application is fully developed. Additionally, packages are managed through inclusion as dependencies within the “package.json” file, which also includes the runtime scripts to execute the application in a development environment or full-scale production. A “vite.config.ts” file contains the local development environment plugin configuration for the application; it declares TailwindCSS, React Router, and logging Terminal modules as plugins for the application. A “react-router.config.ts” file similarly declares necessary configurations for the React Router framework, such as server-side rendering. A “.env” file is included to specify the API url – made to easily reconfigure API requests upon deployment of the back-end API interface – as well as modifications to the Node.js runtime environment to circumvent an issue detailed later in this document. The “tsconfig.json” file contains the configuration of the TypeScript package, which is utilized to enforce the type safety of the application. Additionally, the “Dockerfile” file exists to implement the logic required to containerize the front-end application for a full deployment. It contains declarations for production, development, start and build run options with their dependencies, copying the application directory (in every instance) as well as the “package.json” and “package-lock.json” files (for production dependencies) to the container. The “.dockerignore” ensures that any such containerized solution ignores the large “node\_modules” folder as well as the “.react-router” folders, which exist only to build the application.

## Figure 4.

A screenshot of a computer

AI-generated content may be incorrect.

The front-end file structure. The app folder contains the majority of the application, with the root holding configuration files.

To implement the front-end design, an app folder was created to contain the components and routes detailed in the design. The head of the application exists within the “root.tsx” file, which serves as the entry point for the entire application. The default React app function is declared and executed from here; the app function is implemented to only render routes at a child outlet component with no additional HTML. The “root.tsx” file also defines a layout function and error boundary function to handle the shape of the React framework application as well as requests for nonexistent route pages, respectively. These functions were not modified from the project initialization, as they define industry standard React.js front-end configurations and nonexistent route navigation handling, respectively. The “root.tsx” file is accompanied by the “App.css” file, the only instance of file-specific CSS formatting within the program. This file has been tweaked to provide the application a grey background. All other instances of CSS usage in the front-end utilize TailwindCSS declarations within each corresponding className HTML property to format each HTML component. Within the app folder exists an “api.ts” file that creates and configures the Axios API instance for the front-end runtime to utilize the existing JWT access token as a header for all API calls. The implementation of the default headers defined in this file; however, the header declaration failed to propagate to each instance where the API instance is called throughout the front-end application. The “constants.ts” file declares the access and refresh token strings to use in tandem with getting items from local storage on the client-side device. The routes folder contains corresponding files for the home, login, logout, about, register, and account pages. Additionally, this routes folder contains the implementation for the protected route functionality.

One of the most crucial files within the route folder is the “ProtectedRoute.tsx” file. This file declared login to handle the protection of any nested route from access by a user who is not authenticated with a JWT token. The protected route component maintains authorization through a Boolean function that determines the presence and freshness of the access token by checking the token’s nested expiration date timestamp. If the access token is not fresh and the refresh token exists, a GET API method is executed utilizing the refresh token to acquire a new access token, which is stored on the client-side within the user’s local storage. If the refresh token is not present, the protected route component will automatically redirect the user to the login page; this occurs when the user attempts to access a protected route with both tokens expired or no authentication tokens present in their local storage. Otherwise, the protected route component simply renders any child routes immediately.

Regarding the other routes, the home route file simply renders the welcome component located in the welcome folder. Additionally, the home route conducts client-side loading to discover the presence of a JWT for an authenticated user and passes it to the welcome component to adjust the rendering of elements within the welcome component. The login route file only renders the form component, passing to it props corresponding to the method (login) and the API route. Similarly, the register route file does the same, only adjusting the passed props to the rendered component to adjust the functionality of the form component. The logout route file contains a protected route that clears the local storage, removes the JWT access and refresh tokens before redirecting the user back to the login page. The about route file implements a simple page with pure text and image content that details some of the scope of the application; the content does not change with user authentication. Finally, the account route file contains another protected route that displays the user’s associated job submissions and their results. The route also embeds the align form components to incorporate job submission functionality.

For the components, the welcome component file simply implements the rendered text for the landing page. Depending on the existence of the token – which is supplied via the parent home route – the welcome page will render a different message to an unauthenticated and authenticated user. The form component file declares the rendering of the submission form for login and registration tasks. Taking a property from the parent, it adjusts the visible form fields and API endpoint to submit requests to depending on the method prop. The form redirects the user to the dashboard upon successful login or to the login page upon successful registration. The align form component file declares similar logic, though it does not utilize props passed from the parent route functions. It instead implements a simple submission form that users may provide the unaligned sequences in FASTA format as well as a name for the job. When submitted, the align form component runs the submission function to make the API POST request to the backend via the Axios instance.

The back-end database server is hosted on an external solid-state drive in a dedicated folder for the bioplatform application. This PostgreSQL server has environment variables configured to allow for easy launching from the command line on the deployment hardware, though the nature of the external drive necessitates the adjustment of the associated environment variable to conform to the drive’s assigned letter. The Django application commits “migrations” to the PostgreSQL database to establish the tables and relations necessary for the runtime of the application; these relations only change when the models within the Django back-end application are altered, though they can be accessed utilizing a command line shell into the specific database for the program.

## Figure 5.

A screenshot of a computer program

AI-generated content may be incorrect.

The back-end file structure. General Djangorestframework and JWT administrative files are kept in the root, while the application specific API endpoints exist within their own

For the back-end implementation, a root backend directory is created to hold the application. This root directory contains the “requirements.txt” file which details the required packages to run the back-end Django application. Additionally, a single Python code file named “manage.py” exists at this root as well. This file is the cornerstone of the Django application; running it as the main file via python from the command line allows the user to execute a plethora of subcommands for the purposes of managing the Django application, as well as Django extensions like Djangorestframework and Djangorestframework-simplejwt. This file is utilized to create the database and back-end superuser and create the different migration python files located in “api\migrations.” These files are automatically generated upon the execution of the “python manage.py makemigrations” command from the back-end directory, and they define the methods by which the connected database schema, tables, relations, and other features should be updated upon the execution of the “python manage.py makemigrations” command. This “manage.py” file is also the method by which the back-end server is run; it is essential to the Django implementation.

Within the backend directory exists two other primary directories: api and backend. The second layer backend directory holds the administrative methods. The “urls.py” file located in this folder implements the admin API endpoint that exposes the user models as well as the JWT API endpoints to authenticate users. Additionally, the Djangorestframework provided user method for registration is contained within the same file alongside default API endpoints to access the user and group creation views. The serialization of the supplied generic user model – which is normally contained in a dedicated “serializer.py file – is also implemented in this file for these generic API endpoints. The API uses a different serializer to create users. This backend subfolder “urls.py” file incorporates a URL pattern that implements the underlying API endpoints by including the API’s own “urls.py” file located in the api subfolder. The “settings.py” file contains the configuration for the Django application, which includes the installed applications, any middleware necessary for the back-end runtime (such as the CORS headers), database connection configuration, authentication policies, media directories, languages, and the secret key for production usage. Some additional default configurations are included within the “asgi.py” and “wsgi.py” files for the Django runtime; these were not altered from their vanilla configurations upon project initialization.

Within the api folder exists the application specific implementations and functionality for MSA job creation. The application is defined within the “apps.py” file as “api” to include within the “backend\settings.py” file for proper incorporation into the Django runtime environment. The “models.py” file declares the Job relation defined to manage MSA submissions from the end user. The Job model is composed of name, unaligned, aligned, and phylo text fields, as well as a hidden ID number and an author foreign key connected to the user model. Some timestamp fields also exist on the Job relation. The string return for the model is overloaded to return the unaligned sequence.

The “serializers.py” file implements three serializers: UserSerializer, GroupSerializer, and JobSerializer. GroupSerializer remains unutilized but defines the proper JSON fields for a group that a user would belong to. The UserSerializer constructs the valid user JSON object using the Djangorestframework user generic, though it ensures that the password field is not returned on get by labeling it as write only. Similarly, the JobSerializer labels the author, aligned, and phylo fields as read only to ensure that the end users may not submit these fields to the API endpoint for creation; instead, these fields are populated by the back-end after job processing.

The “views.py” file implements the views available to the end user, constructing the logic behind the API requests. Some default views of UserViewSet, CreateUserView, and GroupViewSet are implemented to allow for user account viewing, user account creation, and user group viewing, respectively. For the runtime of the application, the JobListCreate, JobDelete, and JobDetail views are implemented. JobDelete and JobDetail are simple generic methods that delete and display a Job given the ID, respectively. The JobListCreate view, however, allows a user to fetch all jobs they have executed that still exist within the database. Additionally, the JobListCreate’s POST functionality has been overloaded to extract the user’s submitted sequences and job name from the serializer, before creating a subprocess to execute the MSA. Using temporary files labeled with the job name, the MSA results are read from the alignment file, and the tree structure is read from a produced Newick tree file into an ETE3 tree structure. This tree is then utilized to generate an image file that is directly provided to the front-end application in its public folder, upon which the path to the file is saved as the phylo field in the database. This allows easy loading of the associated phylogenetic tree on the front-end on systems where the entire full-stack application is located; on a proper production environment, this would save a link to the service where the image is hosted for front-end display. These files implement the Django standard to create the back-end processing unit and implement the proposed back-end designed detailed previously.

# Testing and Integration

The application is designed and planned to be deployable to cloud computing solutions. Specifically, any big data solution database capable of connecting to the Django framework is suitable as a backend data storage solution. Additionally, the front-end application contains a Docker configuration file to support containerization and running on cloud computing solutions as well. The only signification deployment limitation is as previously mentioned, where the device hosting the back-end API implementation must be capable of running MSA algorithms as subprocesses, which necessitates individual installations of each MSA. The future enhancements section details a potential solution to this additional back-end hardware requirement. Like the front-end component, the back-end application contains connection configurations to hook into a provided database server, which can be hosted locally or remotely. This allows for easy use of software solutions such as Amazon S3 buckets to serve as the host environment for the database server implementation. Initially, testing methods were to be executed using GitHub actions to create a continuous integration and continuous deployment (CI/CD) pipeline. The program was to use this strategy due to the ease with which frequently employed test cases can be executed and maintained.

Regarding the execution of testing, the application is unfortunately still only tested on a single device at this time. Additionally, the GitHub actions are not currently configured to create a proper CI/CD pipeline; this is mostly due to time constraints. Tests were instead executed via developer discretion in real time. For each requirement of the program, the functionality was constructed and subsequently tested to gauge the performance of the use case. If insufficient, the debugging tools offered by the JetBrains IDEs were utilized to trace back the errors and view runtime data. Solutions were then tested and incorporated accordingly. Additionally, the functionality of many of the API endpoints regarding jobs was subject to the implementation of forms on the front-end application, as these endpoints required authentication to access as well as the properly formatted bioinformatics data. For these instances, the entire full-stack application was run to access the shielded endpoints, which were tested using generically and freely available bioinformatics nucleotide datasets in the requisite FASTA format. Utilizing both IDE’s strong debugging runtimes alongside checkpoint flags within the code, issues were found, isolated, and promptly rectified during testing cycles.

For deployment, the application has only been successfully deployed on a single device detailed in the runtime and development environments above. The front-end, as previously mentioned, has been configured to be suitable for a Docker container, which can then be run on an enterprise solution. However, the actual deployment of the full-stack components has not been completed due to financial hurdles and time restrictions. The enhancements section provides ideas for future deployments in more detail. Currently, all code is hosted on GitHub freely available for cloning/download.

## Difficulties and Limitations

One unexpected limitation when constructing the application was the handling of rendered elements within the React.js environment. Due to the potential size of the data in each job dataset, the resulting alignments may be extremely large; to combat this, the application was to implement a dedicated view page for each job. The entries for each job on the user account page were to be the job title and the accompanying alignments, and upon hitting “show/view,” the end user would navigate to a page populated with the remaining information (unaligned and aligned sequences at this time) corresponding to the job, as well as the name and tree image. However, the handling of rendering within the React environment caused issues persisting the information from the job onto the following page due to state webhooks resetting the values.

Another strange difficulty arose when running the full-stack application and fetching user jobs on their account page. Due to security concerns, Node.js has restricted the maximum size of HTTP headers to avoid denial of service attacks. However, this HTTP request was simply a GET API response corresponding to the request for account job information during client-side rendering of the account page. It appeared that the size of the response header was scaling with the size of the user’s job set due to the size of the job entities, and these header sizes were triggering the security measures from Node.js. Despite an otherwise successful GET request, the front-end application was unable to access the job list. I could not find sufficient information as to why this occurs; therefore, I was forced to adjust the maximum HTTP header size to suppress this problem for now and properly access the GET responses from the back-end.

Additionally, a small issue regarding the implementation of the Axios API configuration occurred. The application is developed to provide an API configuration file within the app folder. This file creates the API instance for the application and declares that every request should utilize the JWT access token as a header. However, upon execution of API requests in front-end components, the requests were not being submitted with the configured header. This was a simple fix of declaring the headers for each API call individually, but further investigation into the failing of this configuration is warranted to avoid code duplication.

For the existing runtime of the environment, only the ClustalW2 alignment algorithm is supported at this time. The ClustalW algorithm produces both an aligned sequence file as well as a DND file containing a Newick formatting tree. This Newick tree corresponds to the aligned sequences, and it can be instantly read into a corresponding ETE3 Tree data structure, which is rendered into an image for display to the end user. The decision to provide only a single MSA technique for the end user was made mostly due to time constraints resulting from earlier difficulties; however, the implementation of the MSA runtime is such that simple adjustments can be made to the front-end as well as the POST API endpoint to submit and handle requests for alternative algorithms.

When attempting to incorporate a deep learning MSA technique, several issues were discovered within the proposed options for the program. The initial algorithm of BetaAlign was revealed to have utilized simulated data for the purpose of training the model. Though the paper reported on the confidence of the nucleotide data regarding the correlation to the traditional “tree of life,” the lack of execution on a proper real-world dataset greatly increased the hesitance to utilize the algorithm for the full-stack application. Additionally, the runtime and reproducibility of BetaAlign’s algorithm was poor due to insufficient documentation, requiring additional analysis of the provided source code to determine additional packages and environment configurations necessary to construct the model. Notably, the provided datasets utilized in the construction of the paper’s model did not conform to general fairseq NLP transformer input files, requiring significant amounts of adjustment to be fed into the fairseq model training methods. Finally, BetaAlign inherited the limitations of fairseq NLPs regarding input size, capping the potential input sentence size to 1024 tokens. The token sizes above this reported drastic performance hits, proving impossible to run. Though MSA datasets can be diverse, containing sequences of varying lengths, this limitation is crippling, as it prevents larger scale gene and amino acid sequence analysis, such as genome-to-genome alignment. This limitation undermines the flexibility of the MSA as a tool for diverse datasets. For these reasons, BetaAlign’s code was not incorporated into the remote repository, but still exists as on the local machine.

For the alternative MSA technique DPAMSA, the initially proposed source code lacked the ability to verify the results and performance reported in the publication. The dataset utilized in the publication was not included with the source code, despite the claims of the authors that it was (Liu et al., 2023). Without reproducibility, I was hesitant to utilize the DPMSA method despite the promising results. Forks of the original DPAMSA source code were also considered due to incorporating their datasets and having overall better supportability for their results. However, issues regarding the implementation of the GA-DPAMSA arose when it demonstrated similar limitations on the size of the input sequences, limiting each sequence to only 8 nucleotide or amino acid units per sequence. This rendered it similarly unsuitable for inclusion as a possible MSA method.

Adding explicability to these deep learning methods proved to be beyond the scope of this project. The nature of the decision-making process in these deep learning architectures is too complex to sufficiently model at this time. Currently, the tools utilized to impart transparency and explainability are insufficient for deep learning architectures. Deep learning architectures are also inherently too complex for direct human analysis into their decision-making processes; we cannot simply log the runtime mathematical learning of the model and understand the construction of the conclusions.

Finally, this report was initially designed to be constructed using LaTeX, a powerful scientific document construction tool with freely available packages to create and display figures, tables, and other graphics. However, the formatting requirements for the final report necessitated a pivot to Microsoft Word as the text editor for this project, where many of the necessary formatting requirements are either default options or readily available to propagate across the entire document. Additionally, the report did not necessitate the inclusion of figures beyond depictions of the full-stack application components, rendering LaTeX’s strong graphical visualizations unnecessary for the task.

# Installation Instructions

To install the application, acquire the source code from the GitHub repository located at <https://github.com/Baltharaaz/bioplat>. Ensure that the device, if running both front-end and back-end components, has Python3 and Node.js installed. Once the source code is acquired, install the packages for both the front-end and back-end applications via “npm install” in the front-end directory and “python -m pip install -r requirements.txt” in the back-end directory. Additionally, the back-end requires that each MSA algorithm option be installed as a separate command line capable executable; at this time, this is only ClustalW2. Please install the command line version of ClustalW and update the PATH environment variable to include the executable so the backend may execute the MSA jobs accordingly. Additionally, the back-end device must have an existing PostgreSQL server installed for the usage by the API. Alternatively, the configuration in “settings.py” file can be altered to connect to a remote PostgreSQL server accordingly.

# Operating Instructions

To launch the back-end application, ensure that the PostgreSQL database server the back-end API is configured to connect to is already running, and launch the server if otherwise. Then, please run “python manage.py startserver” within the back-end directory. To manage any user accounts and database information, it is assumed that the administrator has access to the database server content – whether through a command line or other general user interface – for general query usage. The application is configured to utilize PostgreSQL and, therefore, SQL queries to manage database data from the database shell. However, if the administrator opted for an alternative database, please utilize the appropriate database shell to interact with and configure user and job records. To modify users using the Djangorestframework admin interface instead, ensure that a superuser is created on the host device using the “python manage.py createsuperuser” command on the command line. Navigate to localhost:8000/admin in a web browser of your choice, and log in using the created superuser. The users and groups should now be freely accessible and modifiable from this interface.

For the front-end operation, navigate to the front-end folder and run “npm run dev” to launch the front-end web interface. Ensure that the back-end application is running before navigating to localhost:5173 on the device running the front-end application. In a full deployment, this front-end web application would be associated with a proper URL. You can navigate across the application using the top navigation bar, which leads to the associated pages; navigate back to the dashboard using the hamburger icon. Register an account and log in (or log in using a created administrator/superuser) to obtain access to the account page. From the account page, you can submit your desired MSA sequences in FASTA file format to the right-side input form along with a title. Any results will populate the left side of the screen, though it may require a reload of the account page. With existing jobs associated with your account, the account page will also offer an accompanying delete button for each entry, which will purge the record from the back-end database. You may need to refresh to prompt the account page to reload and remove the record. As a logged in user, you may log out using the navigation bar log out option. Additionally, the landing page adjusts to accommodate the existence of user authentication; at this time, this adjustment is only a message change stating the user has access to the platform.

# Recommendations for Enhancement

I have identified several areas for considerable improvement of the application. For the back-end environment, the development and addition of a specialized deep learning MSA approach would be ideal, though it ultimately was beyond the capabilities and scope of this application. To accomplish this, additional research into the deep learning MSA space should be conducted to potentially select and incorporate a proper existing solution that avoids the limitations of existing deep learning MSA approaches. If that fails, research into the correct deep learning neural network architecture should be conducted. Once chosen, that architecture should be extensively tested and trained on real world biological data cultivated by domain experts to ensure applicability to real life worldsets.

Similarly, the application can still incorporate the DPAMSA and BetaAlign algorithms as possible options for a multiple sequence alignment. Their inclusion, however, necessitates the informing of the end user of the limitations of each MSA technique to ensure that correct datasets are submitted for these algorithms. Additionally, these deep learning technologies require additional investigation as to the application of phylogenetic tree generation and other analytical techniques to maintain the robustness of the application’s results.

New methods in the deep learning multiple sequence alignment appear to be emerging. A recent preprint paper by Arsic and Mayer proposes a convolutional transformer neural network for sequence alignments (2025). Though the paper has yet to be peer reviewed or published, the proposal seems promising for the deep learning MSA subspace. With some investigation and testing of the proposed model, the application could potentially include it as a viable MSA option for users.

As mentioned in the difficulties section, improvements to the result visualization are still an important target. The creation of a dedicated job view page is an important step to ensure end users can properly view the information they possess in the database. Currently, the job view page exists within the code, but the method for passing the information from the account page to the job view is incomplete. Additional investigations into the React.js framework and how the webhooks interact with HTML DOM elements are required to debug this issues and improve the user data view experience.

The user interface also requires the incorporation of a feedback page to allow for user submissions and future improvement suggestions. This addition is not as high a priority, though it is a simple task of exposing form elements to the end user on a dedicated feedback page. Additionally, this will require the construction of a back-end Review model that contains the data submitted by end users; this data would be easily viewable by administrators through back-end tools like the PostgreSQL shell and the creation of Django endpoints to query the new model.

Currently, the back-end API runs ClustalW through a subprocesses creation with Python per the instructions of the Biopy module and common coding practices of bioinformatics professionals. However, the Biopy module includes some MSA techniques within its runtime that have been deprecated. The application should probably be reconfigured to execute all of the included MSA methods within the Python code to avoid the existing reliance on the installation of additional executables. This will require investigation into the existence of Python implementations of the target MSA algorithms outside of the Biopy module. Accomplishing this improvement would allow the application to be easily hosted on cloud or containerized solutions without additional environment configurations, as well as more thoroughly future proof the application through less reliance on exterior executable.

Additionally, the front-end stylization requires more adjustment to be easily readable and aesthetically pleasing. The application should incorporate both a light theme and dark theme configuration for accessibility purposes. Additionally, refactoring the code to be mobile compatible is a strong desire to expand the potential userbase of the application, as the full-stack design was proposed to accommodate those bioinformatics analysts who lack the necessary hardware to execute their tasks.

The application full-stack should potentially implement and incorporate in-application administrative functions. Rather than rely on the Djangorestframework’s exposed administration API endpoint and PostgreSQL’s shell interface, the application should be reconfigured to provide administrative users with an easy interface to handle full-stack data manipulation and management. This ensures that administrators need not be familiar with specific tools to access database data or other domain elements. It also ensures that the only data exposure that occurs to administrative users of the platform is tightly controlled.

Finally, the application must be deployed to a proper cloud environment. This is to ensure that the application is suitable for general user access and utilization, as well as to demonstrate the viability of the application in a production space. Some additional modifications to included environment files and database connection configurations are necessary to accomplish this, but the skeleton of the production implementation exists in the form of the Django back-end’s “setting.py” file database configuration, and the “Dockerfile” on the front-end possessing the methods for containerization. Exploration of existing cloud solutions should be conducted to determine the most suitable locations for both the front-end container and the back-end storage solution.

# Bibliography

Abramson, J., Adler, J., Dunger, J. *et al.* Accurate structure prediction of biomolecular interactions with AlphaFold 3. *Nature* **630**, 493–500 (2024). https://doi.org/10.1038/s41586-024-07487-w

Arsic, P., & Mayer, C. (2025). Ali-U-Net: A convolutional transformer neural net for multiple sequence alignment of DNA sequences. A proof of concept. *bioRxiv*. https://doi.org/10.1101/2025.02.26.640343

Burgstaller-Muehlbacher, S., Crotty, S. M., Schmidt, H. A., Reden, F., Drucks, T., & von Haeseler, A. (2023). Modelrevelator: Fast phylogenetic model estimation via deep learning. *Molecular Phylogenetics and Evolution*, 188, 107905. https://doi.org/10.1016/j.ympev.2023.107905

Dotan, E., Wygoda, E., Ecker, N., Alburquerque, M., Avram, O., Belinkov, Y., & Pupko, T. (2025). BetaAlign: A deep learning approach for multiple sequence alignment. *Bioinformatics*, *41*(1). https://doi.org/10.1093/bioinformatics/btaf009

Jiang, Y., Yang, M., Wang, S., Li, X., & Sun, Y. (2020). Emerging role of deep learning‐based artificial intelligence in tumor pathology. *Cancer Communications*, 40(4), 154–166. https://doi.org/10.1002/cac2.12012

Larkin, M. A., Blackshields, G., Brown, N. P., Chenna, R., McGettigan, P. A., McWilliam, H., Valentin, F., Wallace, I. M., Wilm, A., Lopez, R., Thompson, J. D., Gibson, T. J., & Higgins, D. G. (2007). Clustal W and Clustal X version 2.0. *Bioinformatics*, *23*(21), 2947–2948. https://doi.org/10.1093/bioinformatics/btm404

Liu, Y., Yuan, H., Zhang, Q., Wang, Z., Xiong, S., Wen, N., & Zhang, Y. (2023). Multiple sequence alignment based on deep reinforcement learning with self-attention and positional encoding. *Bioinformatics*, *39*(11). https://doi.org/10.1093/bioinformatics/btad636

Ribeiro, M., Singh, S., & Guestrin, C. (2016). “Why should I trust you?”: Explaining the predictions of any classifier. *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Demonstrations*, 1135–1144. https://doi.org/10.18653/v1/n16-3020

Sapoval, N., Aghazadeh, A., Nute, M. G., Antunes, D. A., Balaji, A., Baraniuk, R., Barberan, C. J., Dannenfelser, R., Dun, C., Edrisi, M., Elworth, R. A., Kille, B., Kyrillidis, A., Nakhleh, L., Wolfe, C. R., Yan, Z., Yao, V., & Treangen, T. J. (2022). Current progress and open challenges for applying deep learning across the biosciences. *Nature Communications*, *13*(1). https://doi.org/10.1038/s41467-022-29268-7

Sun, C., Fang, R., Salemi, M., Prosperi, M., & Rife Magalis, B. (2024). DeepDynaForecast: Phylogenetic-informed graph deep learning for epidemic transmission dynamic prediction. *PLOS Computational Biology*, 20(4). https://doi.org/10.1371/journal.pcbi.1011351

Thompson, J. D., Higgins, D. G., & Gibson, T. J. (1994). CLUSTAL W: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice. *Nucleic acids research*, *22*(22), 4673–4680. https://doi.org/10.1093/nar/22.22.4673

Yousef, M., & Allmer, J. (2023). Deep learning in bioinformatics. *Turkish journal of biology = Turk biyoloji dergisi*, 47(6), 366–382. https://doi.org/10.55730/1300-0152.2671