MASARYK
UNIVERSITY

FACULTY OF INFORMATICS

LLWS: REST backend for centralized execution of bioinformatics calculations

Bachelor's Thesis

PETR KABOUREK

Brno, Spring 2023

MASARYK UNIVERSITY

FACULTY OF INFORMATICS

LLWS: REST backend for centralized execution of bioinformatics calculations

Bachelor's Thesis

PETR KABOUREK

Supervisor: doc. RNDr. Barbora Kozlíková, Ph.D
Supervisor specialist: Mgr. David Bednář, Ph.D

Department of Visual Computing

Brno, Spring 2023

FACULTAS ARTIS INFORMATICÆ • UNIVERSITAS MASARYKIANA •

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Petr Kabourek

Supervisor: doc. RNDr. Barbora Kozlíková, Ph.D
Supervisor specialist: Mgr. David Bednář, Ph.D

Acknowledgements

I would like to thank the entire team of Loschmidt Laboratories for allowing me to work on this project and for support throughout. I would also like to thank Sérgio Manuel Marques, PhD. for proofreading the text of this thesis. Lastly, I would like to thank doc. RNDr. Barbora Kozlíková, Ph.D. and Mgr. David Bednář, Ph.D. for their guidance and their invaluable help with this bachelor thesis.

Abstract

The aim of this work was to develop a new service called Loschmidt Laboratories Web Services (LLWS). Its main purpose is handling computations of protein-engineering web applications developed in Loschmidt Laboratories, MU. LLWS offers a simple approach to implementing bioinformatics tools without the complexity associated with managing calculations on computing clusters. The logic of the calculations itself is fully hidden behind an abstract layer of an easy-to-use interface that allows developers to focus on developing the applications. LLWS is compatible with several environments and at the same time offers the possibility to expand it with new architectures. Another functionality is communication with third-party APIs. The service can fully conceal the differences and API problems of different service providers. LLWS can also convert files with calculation results into easy-to-process JSON format. All functionality is available via HTTP endpoints, which are intended to be accessed using the supplied client library for the Java programming language.

Keywords

bioinformatics, REST backend, HPC

Contents

Introduction **1**

1 Proteins **3**
1.1 Protein characteristics 3
1.2 Protein engineering 5

2 Software tools in the Loschmidt Laboratories **7**
2.1 Developed software 7
2.2 Architecture of the tools 8
2.3 Shortcomings of current architecture 10

3 Laboratories Web Services (LLWS) **12**
3.1 Functional requirements 13
3.2 Non-functional requirements 14
3.3 Architecture 14

4 Implementation **16**
4.1 Task implementation 21
4.2 Facades implementation 25
4.3 Parsers implementation 26

5 Current deployment and Outlook **28**

A An appendix **30**

Bibliography **31**

List of Figures

1.1 Active site marked in green. Bonded ligand is in black colour. (PDB: 9LYZ, Generated using PyMOL program) . 4
1.2 Overview of steps in rational design and directed evolution (taken from: [7]). 6
2.1 Schema of workflow of HotSpot Wizard 2.0 (taken from: [9]). 9
4.1 Diagram of packages and their transitive dependencies on each other. 18
4.2 Available endpoints for user management. Website generated by integrated Swagger-ui package. 20
4.3 Available endpoints related to tasks execution. Website generated by integrated Swagger-ui package. 23

Introduction

Proteins are fundamental constituents of biological systems. Their diverse functions and applications in a variety of industries, such as food production, chemical engineering, and pharmaceutical development, have made them the subject of intense scientific inquiry. With the increasing affordability of computational power, many in silico tools and algorithms have been developed to analyse proteins.

The Loschmidt Laboratories based at the Faculty of Science, Masaryk University, have contributed to this field by developing and providing bioinformatic tools for protein analysis, that are used by thousands of users around the world, most of them web-based. These tools integrate and execute multiple algorithms into complex workflows. Each of these tools implements the execution of underlying algorithms separately from others. This leads to common code duplication. As new applications are developed, maintaining the tools and reacting to changes and API breaking updates of remote services becomes increasingly challenging.

The aim of this thesis is to implement a web-based backend service that creates an abstraction layer over the execution of computational tasks, parsing their output data files, and communicating with APIs of third-party provided services. This service will provide a single place for the implementation of task execution and simplify the process of developing new tools. By centralizing the implementations, maintenance of existing tools will also be simplified, as changes will only need to be made once.

This backend service is called Loschmidt Laboratories Web Services, or LLWS for short. The main features are realised by three concepts. Such are: tasks for the execution of computational tools, facades for communication with the remote API, and parsers for reading output files and interpreting them in JSON format, which may be easily handled using any of the multitude of the existing libraries.

LLWS is implemented with robustness in mind, ensuring that the application can restore its state after a crash or an unexpected restart from the contents of a database and auxiliary locally stored files. It is also designed to filter out potential technical failures, with retrying policies in place to minimize the need for client applications to handle

errors. Tasks are implemented in a modular way, which supports simple addition of new execution infrastructure or framework.

User authentication and storage management are also included in LLWS. Storage-related endpoints are utilized to manage files, and virtual sittings are supported to enable users to organize files according to their needs without interfering with other users and their activities.

LLWS is currently being used by a newly developed web application, PredictSNP Onco [1], and other tools are currently being migrated or will be migrated in the near future.

The thesis is structured in the following way. The second chapter of this thesis provides a basic overview of proteins, their characteristics, and their functions. The following chapter describes the software developed in the Loschmidt laboratories, its architecture, usage, and shortcomings and issues. The fourth chapter explains how these shortcomings would be addressed by LLWS and its functional and non-functional requirements. The fifth chapter details the implementation itself. It describes the interaction from the user perspective as well as the inner logic under the abstraction of HTTP endpoints. Last final chapter discusses the current state and uses of the service and potential future development and plans. The language model ChatGPT optimized certain parts of this thesis to improve clarity and readability.

1. `https://loschmidt.chemi.muni.cz/predictsnp-onco/`

1 Proteins

Proteins are essential macromolecules that play crucial roles in a vast array of functions and processes in all living organisms. These complex molecules are involved in everything from structural support and movement to enzymatic reactions and signalling pathways. Without proteins, life as we know it would not exist[1].

1.1 Protein characteristics

Proteins are made up of one or multiple long chains of amino acids, which are linked together with peptide bonds. There are 20 different amino acids that can be combined in countless ways to create an enormous variety of proteins with different functions and properties. The sequence of amino acids in a protein determines its shape, properties and, ultimately, its function[2].

Proteins with long and fibrous structure, such as keratin and collagen, are usually insoluble in water and therefore they are often found acting as natural building blocks for cells. Other, more compactly packed proteins, such as haemoglobin, act as transport media, carrying oxygen and other essential molecules throughout the body. Still others, such as insulin and other hormones, serve as signalling agents that regulate bodily functions and control ongoing reactions[2].

One of the most important groups of proteins is the enzymes. Enzymes are catalysts that accelerate chemical reactions by lowering the activation energy required for the reaction to occur, meaning that their presence and function are key for the majority of the processes in any organism, starting from the decomposition of other molecules, such as digestion, to its contrary – the synthesis of new molecules, such as DNA replication.

A characteristic feature of all enzymes is the presence of at least one active site (or catalytic site). Example of an active site is shown in Figure 1.1. An active site is usually a cavity within is structure, or a dent on its surface, where the reaction occurs. The active site may be buried within the protein. In this case, it is connected with a tunnel leading outside the protein, that allows the substrate – the molecule

that the enzyme interacts with – or other ligands to enter or to exit the active site.

The spatial arrangement of the amino acids around the active site are essential for the enzyme function. Any minor change in this place may significantly affect its ability to perform a specific reaction. In many cases, a change will be detrimental to the catalytic activity, rendering it completely inactive. However, in certain instances, changes in the protein structure may increase its efficiency and speed of ongoing reactions.

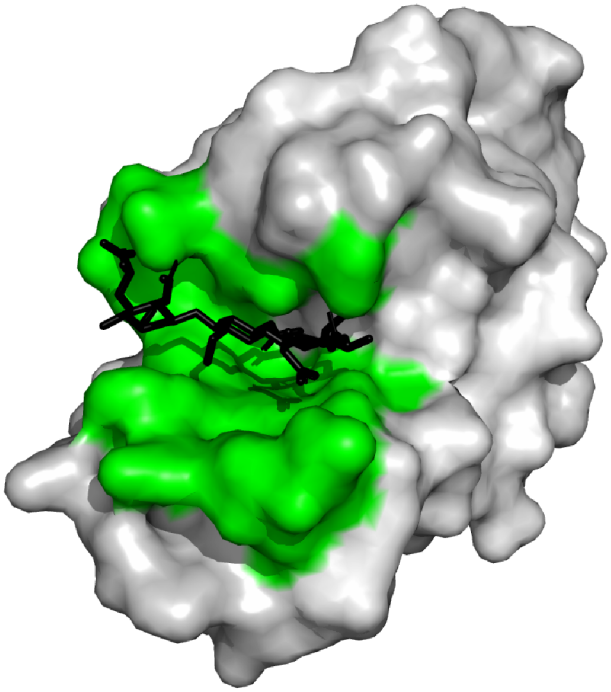

Figure 1.1: Active site marked in green.
Bonded ligand is in black colour.
(PDB: 9LYZ, Generated using PyMOL program)

Unlike other inorganic catalysts, enzymes are often highly specific and typically they only catalyse reactions involving particular substrates or very close relatives. This specificity also arises from the precise arrangement of amino acids around the enzyme active site.

Enzymes also require specific conditions in order to function properly. For example, the pH range and temperature of the environment can have a significant impact on the enzyme activity. Enzymes have an optimal pH range and temperature range at which they function most efficiently. Changes to pH or temperatures outside of this range can significantly affect enzyme activity, and may even cause the enzyme to become denatured and the entire structure collapses, resulting in the complete loss of activity[1].

1.2 Protein engineering

Due to their potent catalytic activity and specificity, enzymes have become indispensable tools in many industries. Their usage spans from food production (brewery, cheese making), all the way to chemical industries such as biofuel, laundry detergents manufacturing, and pharmaceutical industry.

With proteins being commercially used, a new search for augmenting their characteristics has begun. More efficient enzymes, that are stable in harsher conditions are constantly being sought after. Thus, a new field of study emerged – protein engineering[3].

Protein engineering often interconnects bioinformatics and biochemistry to analyse and describe proteins, their structure and functions. The collected data is then used to design new proteins with enhanced properties. It is usually achieved by the mutation of an existing enzyme and replacing certain residues within its chains for different amino acids. Two complementary approaches currently used in protein engineering are the directed evolution and the rational design. The steps of each approach are shown in Figure 1.2.

Directed evolution is an experimental method that works on similar principles as natural evolution does. A large number of randomly mutated proteins is created and screened. Specimens with the best parameters are then selected for the next round of mutagenesis until a protein with the desired properties is found. While directed evolution requires very little knowledge about proteins, it is also incredibly laborious and financially intensive procedure with no guarantee of success[4][5].

Rational design aims to use computational power along mathematical models, algorithms and experimentally collected data to attempt and predict specific mutations for a given protein and purpose, which would result in a new molecule with the intended parameters. Such approach offers much lower time and financial requirements. However, it is heavily dependent on the quality of the provided scientific data, the used mathematical models and other parameters[6][3].

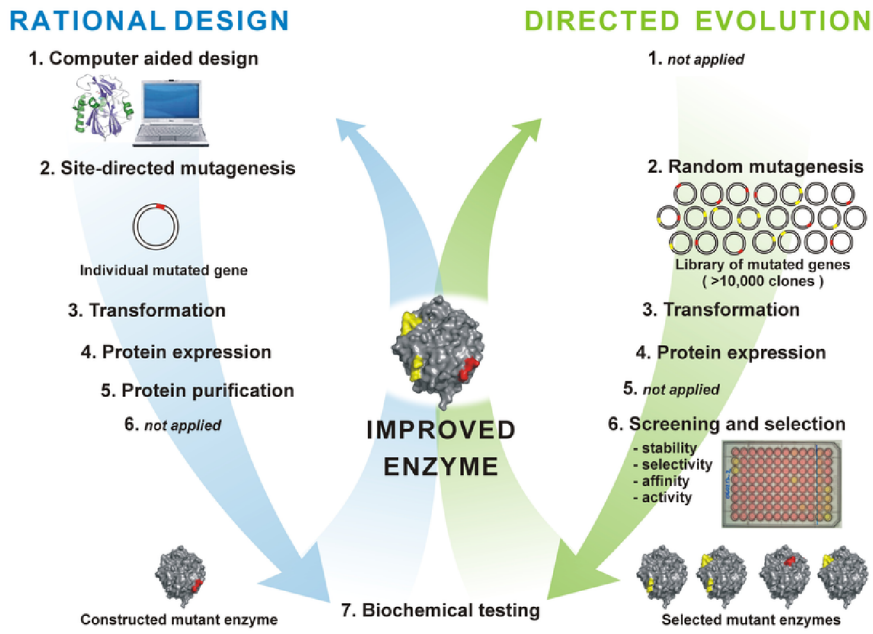

Figure 1.2: Overview of steps in rational design and directed evolution (taken from: [7]).

There are a lot of rational design-based algorithms and programs offering various kinds of information and predictions. These computational tools range in complexity, from the simple static structural analysis of a protein to the more demanding molecular dynamics simulations, which sometimes can run for multiple days or weeks. However, most of these programs are standalone tools, and researchers often need to use multiple post-processing calculations to obtain comprehensive results[8].

2 Software tools in the Loschmidt Laboratories

In the Loschmidt Laboratories, bioinformatics applications are developed with the aim to seamlessly integrate multiple algorithms into a complex workflow. The goal is to provide the users with results that are easy to access and utilize.

All tools are designed as web-based applications that are available online for free of charge to non-commercial users around the globe.

2.1 Developed software

The 6 main computational tools building on previously described architectures that the Loschmidt Laboratories provide are:

HotSpot Wizard 3.0

HotSpot Wizard is a computational tool designed for generating mutation libraries to enhance the stability, catalytic activity, substrate specificity, and enantioselectivity of proteins. It integrates four distinct protein engineering strategies, thereby enabling users to selectively target sites that influence the aforementioned properties[9]. Since its initial deployment in 2016, the tool has been visited by over 79 000 users.

FireProt 2.0

Webserver for predicting multiple-point mutations enhancing protein stability. FireProt integrates sixteen computational tools combining energy and evolution based approaches to suggest mutations with additive stabilizing effects[10]. There are currently 36 500 visitors on record since its release in 2015.

EnzymeMiner 1.0

This application is designed to automatically and periodically mine genomic databases for sequences of potential interest. The server employs a variety of sequential and structural bioinformatics tools to identify protein-coding genes that contain conserved catalytic residues,

which are presumed to be involved in specific enzymatic reactions[11]. The server has been visited by over 17 800 users from the year 2020.

Caver Web 2.0

An interactive web server designed for the automated analysis of tunnels and channels present within protein structures, as well as the transport of substrate molecules into the protein active site[12]. Since its release in 2019, the server has processed over 55 300 jobs and has been utilized by more than 30 000 unique visitors.

FireProt ASR 1.0

FireProt ASR 1.0 focuses on the search and construction of evolutionary trees for a given enzyme with little to no prior knowledge of the protein background[13]. Since its release in 2021, FireProt ASR 1.0 has received nearly 15 500 visitors.

PredictSNP Onco 1.0

PredictSNP Onco 1.0 is the most recently released tool in 2022. It is designed to assess oncologically relevant effects (carcinogenic) on sequential and structural properties of a protein, and apply a virtual screening method to suggest possibly applicable drugs for the treatment[14]

2.2 Architecture of the tools

The tools are composed of two main parts: the user interface and the computational core.

The user interface is responsible for facilitating the communication with users, job submissions, and display of results. Upon the job submission, all input files are compressed into a single tar file, and a new job is recorded to a job database for the specific tool. The user interface then awaits completion of the job. Once the job is finished, the results are downloaded from the computational core and displayed upon request.

The computational core is a monolithic service performing all necessary tasks for the successful completion of a job. Each application has its own standalone core completely independent of other tools. The majority of the computational core is the same for all applications. This common part contains the framework on which all tools function. This includes storage management, database handling as well as email notifications. A crucial feature of the common core is also a dependency system. The workflow of each application is defined by a sequence of calculations, which must be always followed, as calculations may require data from previous steps in the workflow. An example of such workflow is shown in Figure 2.1. Therefore, the queuing system must have been integrated within the computational core, which ensures that no calculation will be executed before all its dependencies are met.

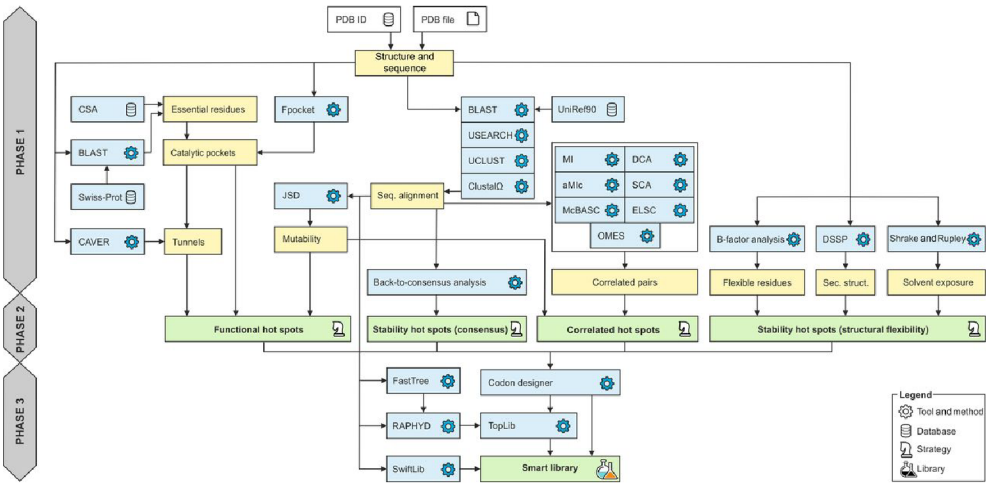

Figure 2.1: Schema of workflow of HotSpot Wizard 2.0 (taken from: [9]).

The shared part of the core is then extended with modules and data structures holding the result data. These extensions are unique for each application. The exact logic of the calculation is implemented here.

Each module typically executes a similar sequence of steps that includes the loading of input data, executing a calculation with a tool,

parsing and modifying the results of a calculation and passing the data to the following modules.

Once a new job appears in the database, the input files are downloaded from the user interface and the computational core begins executing all dependency-free modules, while putting others on hold.

Computations by the tools are executed on a high-performance computing cluster. Currently the only supported platform is PBS. The submission of a job to PBS is also within the responsibility of each module as well as monitoring the execution status and handling possible errors.

Once a module of a job is finished, the modules waiting for dependencies are checked to determine if their dependencies have been met, and if so, they are executed. This cycle repeats until all modules are finished. Once completed, the database is updated and the results are packed and stored until they are requested by the user interface. In the event that any module fails or encounters an error, the entire job is cancelled and marked as failed without any retrial.

2.3 Shortcomings of current architecture

The current architecture of the computational core was originally designed for a single standalone application, which led to certain problems when it was employed by multiple other applications.

Firstly, the current implementation causes inability to operate outside specific PBS clusters. Computing on a new PBS cluster requires all the necessary software to be installed on said cluster, which causes any theoretical migration to be very time consuming and complicated. Additionally, the PBS technology is becoming outdated and is being pushed out of favour by newer high-performance computing systems such as Kubernetes or Slurm.

Secondly, modules implementing their own submission and monitoring logic create additional work on the developers. Instead of only having to deal with a workflow that is biologically relevant, accurate and reliable, it is necessary to also focus on the technical part, ensuring a proper communication with the PBS cluster and error handling.

Thirdly, the calculations used within the workflow are often shared among multiple applications. As an example, BLAST search, which

statistically evaluates sequences[15], is executed by HotSpot Wizard, FireProt, EnzymeMiner, and FireProt ASR as well. AutoDock Vina, attempting to predict the optimal position of a ligand being bound to an enzyme[16], is used in Caver Web, HotSpot wizard and PredictSNP Onco. However, as the implementations of modules are not shared among the different applications in any way, each is implemented slightly differently, and more often than not, by completely different people. This results in a needlessly time-consuming upkeep and maintenance process for the source codes.

This issue is crucial, especially when modules do not submit to the PBS clusters, but rather contact a remote third-party API, for example, while contacting and searching databases. When the API is modified, each application using the said API has to be updated to reflect this change separately.

Lastly, modules running computations are implemented to fail on any error with no attempt to retry. This relies on computations running as reliably as possible, which is difficult on a PBS cluster to achieve. PBS is mainly a scheduling system, therefore, when a computation is running on a node, any issue on this node will cause the computation to fail. PBS issues are the most common reasons for job failures in the Loschmidt Laboratories. This results in the users' job failing due to no error on their side. This negative experience may result in the users not using the application again because of reliability issues, especially if a job runs for tens of hours only to fail. A more reliable and robust error-handling of the possible failures at the computational node level would severely reduce the failure rate of all currently offered applications by the Loschmidt Laboratories.

3 Laboratories Web Services (LLWS)

As new tools are developed, the aforementioned limitations are becoming increasingly apparent, prompting the search for alternatives to currently used implementation of the computational core.

After experimenting with the existing workflows managers and executors, the idea of reimplementing the entire core into Common Workflow Language (CWL) within an existing framework and building on top of already existing features[17] was ultimately dismissed.

The two candidates, that were considered the most, are the combination of Workflow Execution Service (WES)[18] and the Task Execution Service for Kubernetes (TESK), both developed by Elixir, and Steep workflow manager[19].

WES is a workflow manager that can also submit calculations to Kubernetes using TESK service. A prototype pipeline of FireProt ASR was successfully created using only workflows defined in WES. However, WES is currently incapable of creating loops or indeterminate number of tasks on the fly. This feature is necessary for multiple applications in the Loschmidt Laboratories and there was no way found to circumvent this requirement.

The second Workflow manager considered was Steep[19]. Steep has the capacity of creating cyclic workflows and adjusting the workflow based on the calculated data. While Steep provides a robust, reliable and agile way of executing workflows, its agent-based execution effectively disallows the usage of PBS or another external queuing system.[19][20] Developing with Steep would require changing completely the production environment architecture and environment. It would also necessitate the complete reimplementation of all applications from scratch. This was deemed too demanding for the small development team at the Loschmidt Laboratories.

As a more manageable solution to this problem, an approach of further splitting responsibilities was adopted. A new service would be created, which would fully manage the calculation execution and partake the error-handling. The existing computational core would be left mainly as a workflow manager, instructing the new service which tasks to execute and which parameters to use. This approach allows the execution cores to remain in their current form, with only

the execution logic being removed and reimplemented elsewhere for all applications in a centralized location. This new backend service was named Loschmidt Laboratories Web Service, or in short LLWS.

3.1 Functional requirements

The primary goal of the LLWS is to remove the burden of the execution logic from clients. To achieve this, the service must have the ability to execute computations while completely hiding the platform and technical details from clients. Clients should only be required to provide input data and await results, with the execution process being implemented in a way that makes adding new types of calculations as easy as possible.

To ensure flexibility, the service should be capable of submitting to multiple execution platforms, rather than being limited to a single architecture like PBS. To achieve this, a modular implementation of calculation submission should be in place to allow for easy addition of support for new execution environments or frameworks.

In addition to executing calculations, LLWS should hide the APIs of third-party services and any technical difficulties on their side. The service should offer a uniform way of contacting third-party APIs, allowing for easy submission by client applications.

To minimize the need to reimplement the logic of reading and parsing files in each application core, LLWS should provide the capability to parse output files into a single, easily programmatically readable format. JSON was selected as a modern, lightweight data format that is often used for information transfer between applications.

All of these features should be available via HTTP APIs that support authentication and access control for each endpoint. A system of APIs must work with reliable backwards compatibility, and the introduction of new features cannot require changes to existing client code. This is essential, as only one instance of LLWS is planned to be running for all applications, and it is not feasible to update all applications with changes to LLWS API simultaneously.

3.2 Non-functional requirements

The aim should be to achieve maximum reliability and minimize the need for service maintenance. Specifically, LLWS must be capable of fully recovering its state after an unexpected restart with minimal impact on the running tasks. Additionally, LLWS should be equipped to handle potential technical issues such as network connectivity and automatically recover once the complications are resolved.

A second key requirement is to provide modularity with the types of tasks, third-party APIs and parsers, as the number of tools in the workflows used in the Loschmidt Laboratories is likely to grow in the future. This is why new types of existing features must be easy to implement and extend.

Since all client applications in the Loschmidt Laboratories are implemented in Java, a Java client library should be implemented to facilitate the communication with LLWS.

3.3 Architecture

The entire system is comprised of three parts: the standalone application, a Python auxiliary scripts to manage calculations, and Docker images with the prepared software.

Application

The application contains the main business logic of the service and heavily relies on a database that stores its entire state. To ensure the full state recovery after a restart, no key information is stored in runtime. Clients communicate exclusively with the application, and all communication and management are based on HTTP endpoints that require authentication by an account with the appropriate privileges.

Python library

The auxiliary Python library is designed as a package of scripts that handle the execution logic of each task during task execution on the HPC. Its purpose is to download and manage input files, execute the

designated commands for a specific calculation, extract and upload the resulting output files, and collect the logs of completed calculations.

Docker images

To make the execution of tasks as environment-independent as possible, a Docker container is used. As the final part of this project, a collection of Docker images was created, each specific to a particular task. Each image contains all the necessary software and the LLWS Python library with the execution scripts. For execution, the appropriate script is selected and then it is given control of the execution within the container. For cases like the PBS nodes, or other similar environments where the users are not typically given permissions to run Docker, these images may be recreated as singularity[21] files to avoid requiring root access.

4 Implementation

As all web applications developed in the Loschmidt laboratories are written in Java, LLWS was also implemented using the Java programming language. It is built on top of the Spring Boot framework[22], heavily interacting with PostreSQL[23] as a database provider.

Spring Boot was chosen because it is an enterprise-level standard for developing web-based applications and microservices. Due to its wide-ranging libraries and tools that can be easily integrated into the application. This includes tools such as Spring Data for database access, Spring Security for authentication and authorization and other libraries for faster application development.

The Spring framework employs the concept of Beans, which refer to objects that are created and managed by the framework. This allows for streamlined management of object creation and linking between them, relieving the programmer of manually handling these tasks. Instead, the programmer can focus solely on specifying the dependencies of each Bean, which the framework will then handle automatically.

PostgreSQL was chosen as an open-source database well known for its robustness, reliability, and extensibility. However, the only database specific feature used, is the concept of a schema. Therefore, the database provider can be relatively easily replaced by an other provider supported by Spring Boot.

The project is divided into Maven submodules to clearly divide the responsibilities between the different parts of the code, and potentially allow multiple developers to work on specific parts of the application simultaneously.

The Figure 4.1 illustrates the mutual transitive dependencies among the submodules. The blue and green elements in the figure represent key components of the system. All yellow packages are meant to be compiled and used toghether and are unlikely to be usefull as a standalone package. The green package, namely llws-server, is a fully functional executable application that is ready for deployment. On the other hand, the blue packages refer to client libraries designed to be utilized by client programs. The submodules and their funtionalities are:

llws-api Defines the fundamentals of communication between clients and the applictaion.

llws-client Contains client classes to be used by other software interacting with LLWS.

llws-facades Implements requests and execution of third-party APIs.

llws-mailer Allows the application to send email messages, which can be used for monitoring purposes or to contact users at the request of client applications.

llws-pbsm Provides the main module for connecting to PBS via PBSM software.

llws-security Contains implementation of user authentication and role access management.

llws-server Interconnects al other modules and adds heavy Spring-Boot dependencies to produce final executable with the LLWS application.

llws-session Introduces the concept of sessions to offer clients virtual sittings to eliminate conflicts between activities of different users.

llws-storage Adds storage and file management logic.

llws-utils Provides utility methods and classes.

llws-hpc Builds the generic foundation for submitting tasks to High-Performance Computing clusters.

llws-tasks This submodule includes generic calculation management.

llws-parser Implements parsing output files into JSON format.

llws-k8s Allows for submission of calculations into a Kubernetes cluster.

llws-client-spring Extends llws-client with Spring dependencies for client integration into Spring Boot framework applications.

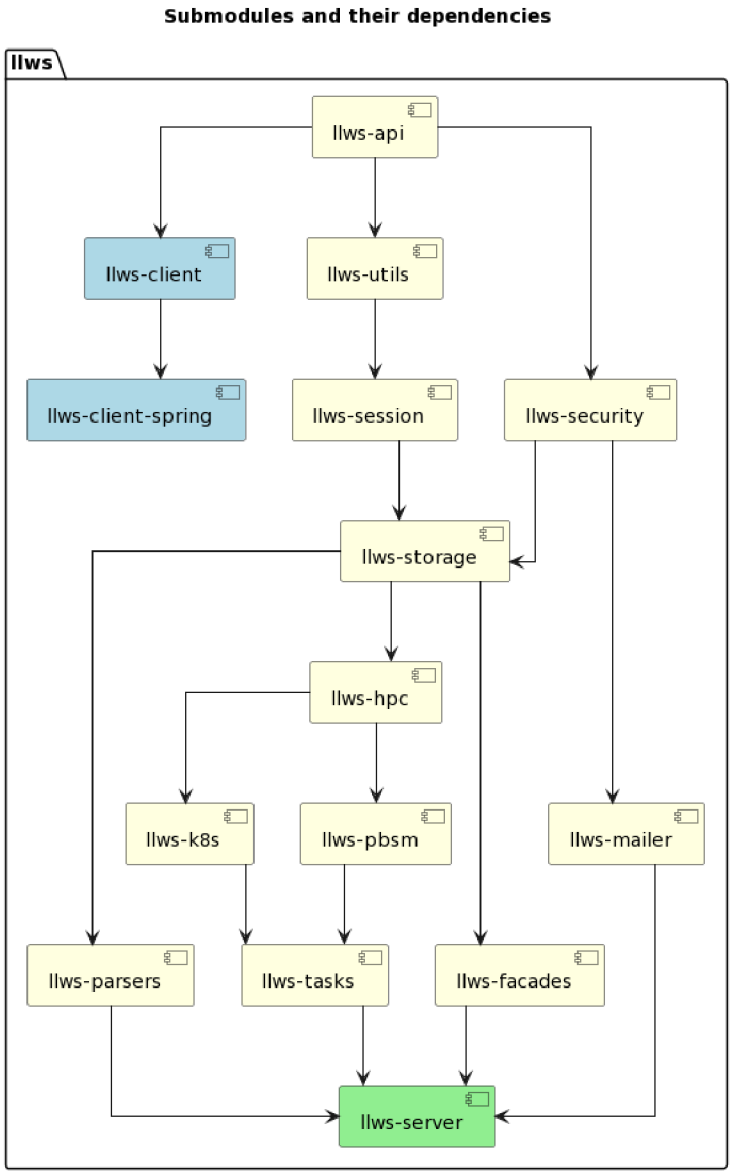

Figure 4.1: Diagram of packages and their transitive dependencies on each other.

The llws-api is integral to most other submodules as it contains specifications of available endpoints of the API. Definition consists of an URL path and classes that define the parameters of each endpoint.

The requests are then automatically mapped to these classes within the Spring framework. In order to ensure backward compatibility, all the endpoints are versioned. This means that if a change needs to be made to an existing endpoint, a new endpoint of a higher version is created and the changes are implemented there. As a result, multiple versions of endpoints can be kept available indefinitely until they are deemed obsolete.

Each version of an endpoint requires its own set of classes. To avoid complications arising from multiple class implementations, every submodule that integrates the llws-api module introduces its own data classes, referred to as domain classes. Data obtained from an endpoint is mapped from endpoint-specific classes into the domain class, which is then used in the internal components of the application.

Both llws-client and spring-client compile into client libraries, which are available as Maven dependencies in private repository of the Loschmidt Laboratories. These libraries are provided as a single client containing all the features related to LLWS. To avoid manually defining JSONs to be sent to LLWS, all input and output JSONs are represented as Java classes which are translated back and forth using Java Jackson library. Most classes in this client provide fluent API, making the code concise and easy to implement without the need to create numerous variables. Additionally, the client provides internal checking of tasks states thus hiding the asynchrounous nature of LLWS APIs. Client aims to return classes from java.util.concurrent library to allow users to decide between blocking or non-blocking waiting without the need to implement additional checks and API communication of state. The translation of code into API calls and requests is handled using Retrofit2[24] library.

Llws-spring-client then allows seamless integration into other Spring Boot applications. It allows configuration of the client via application.properties file, which serves as the primary configuration location for Spring Boot. Furthermore, it creates Beans of the main classes that provide access to the functionality of the client.

Llws-security implements the concepts of a user, role, and privilege. It essentially implements a role-based access authorization system. Each user can log in and access the LLWS API. Each endpoint is bound to a specific privilege, which grants permission to interact with it. Privileges are then grouped together into roles. The user is assigned

appropriate roles to allow access to the selected features. Each user can hold multiple roles. Users and roles are stored and loaded from the database, while privileges are taken from the source code. The application then searches for all privileges in its code and ensures that they are registers in the database. An admin user with an admin role, including all permissions, is hardcoded into the llws-security package. If they are not recorded in the database at the startup, the application creates a record for each. Other user management is available either through the database or via the API, whose enpoints are shown in Figure 4.2.

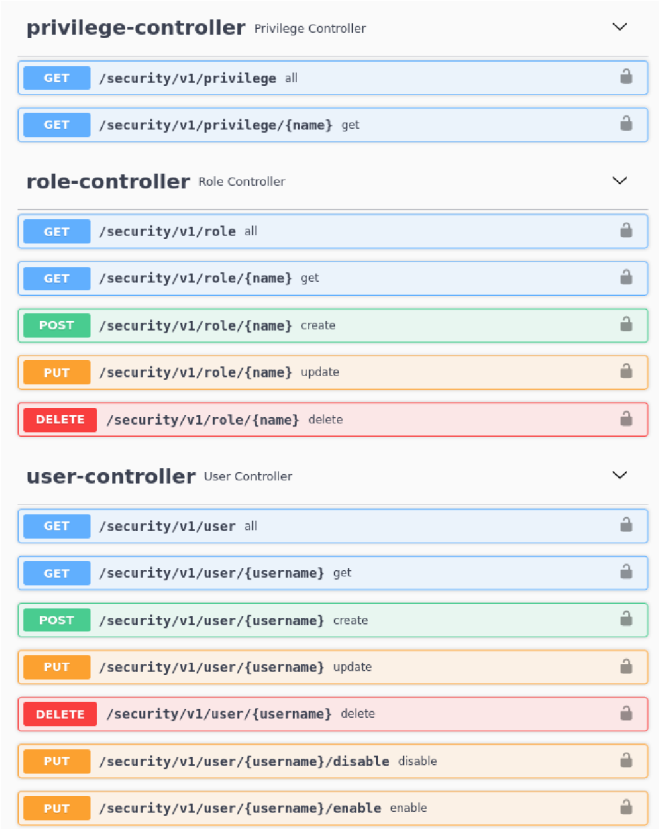

Figure 4.2: Available endpoints for user management. Website generated by integrated Swagger-ui package.

The llws-session module allows users to create virtual sittings. These sittings are implemented to separate data of multiple users

running the same tasks on different data. Any user can create a new session using a dedicated endpoint and is given a session ID, which is then used to upload files to their sitting, download results, and execute other features. Each session is given a directory, where users can organize files according to their needs using the storage API.

The storage API gives users the option to upload and download files as well as handle and manage the directory structure of the uploaded files without the risk of affecting other users and their activities.

The storage available for users physically exists as an S3[25] storage. This solution was chosen because it is a simple way of making any uploaded files available to the application as well as to any execution environment with internet access. Furthermore, S3 is a simple HTTP based protocol and as such it is easy to use programmatically and it requires no complicated firewall policies to be in place. S3 is also recognized as the standard protocol for cloud-native applications, which is a valuable for LLWS, which is being developed to be capable of operating within Kubernetes.

On this storage, a directory with "static" files exists. These files are available for all authenticated users to read and use in the tasks, however they cannot change the files here or upload new ones, as a special privilege is needed for write permissions. This place is meant to hold configuration files, and other small data files that are identical in many task runs. By having such files available in one place, the user does not have to upload them to each virtual sitting and waste space in the storage.

4.1 Task implementation

From the user perspective, the entire task logic is available via a set of endpoints that allow the user to execute a task, check its status and access the results. Internally, each task is defined using three JSON files that are included directly in the source code of the LLWS application in the llws-tasks package. Each task type is identified by a name and version number, which are used to select the JSON files describing the appropriate task.

The first JSON file is a schema used for validating the calculation input parameters provided by the user. When a user calls the submit

endpoint, they provide a JSON including all data for a given task. This includes the task names and the version number. Other section of the request body contains parameters that will be passed to the tool being executed. Before submission, these parameters are verified against the schema, to ensure that only tasks with correct and reasonable inputs will be run. If this check is passed, the parameters are uploaded to the S3 storage to a folder specific for the user session, and the task is submitted.

During the task submission, the internal logic is driven by values saved in second definition JSON file with task metadata. The primary information is the task type. Each task type defines a different execution environment and every type of task has its own executor. Currently, three environments are supported – PBS (llws-pbsm), Kubernetes (llws-k8s) and local execution.

The local executor runs tasks directly on the server where the Java application is running. It is intended for very fast and simple tools not to be slowed down by queues and other overhead of dedicated computational clusters. If the server is restarted, all unfinished local tasks will be rerun. PBS submissions are handled by PBSM – a HTTP based PBS submission service developed in cooperation with the Loschmidt Laboratories. Communication with PBSM is handled by the llws-pbsm package. Kubernetes submission is done natively using fabric8[26] client integrated in llws-k8s.

Additional values in the JSON containing metadata vary based on type of task, as each executor requires different parameters.

The last JSON file contains schema for submission parameters. This is last part of the request body specified by the user. It gives the user an option to modify the resources committed to their task. If an unusually demanding task is being submitted, the RAM, CPU or time limit may be increased to avoid calculation failure due to insufficient resources.

Along with the submission, a database record is created with all crucial information about the task. The user is given an UUID that allows them to track the task further. To monitor the progress of the task, the status endpoint is available. As running tasks are not under control of the LLWS application, and the status of a task is checked upon the user request, even after a restart or a crash the tasks will continue running without interruption. The only exception are the

local tasks, but they are expected to finish quickly, and therefore, after a crash all unfinished local tasks will be restarted.

On the computational node, the docker image created specifically for a given task is loaded. Inside, the execution of the task starts by loading the LLWS Python library. This library downloads the user-entered parameters from the S3 storage along with the input files and then executes the commands for running the algorithms. After the calculation is finished, the script collects and uploads the output files and logs back to S3 storage, where the files will be available for download by the user.

To make the analysis of the task run be more easily readable for the user, the concept of a report is introduced. Each command in the python script is considered to be a step of the execution. The report then consists of basic information about the task and its series of steps. Each step contains information about the time when the given command was executed, when it finished, its return code, and path to standard and error output log files available in the S3 storage.

User can access the tasks using endpoints listed in Figure 4.3.

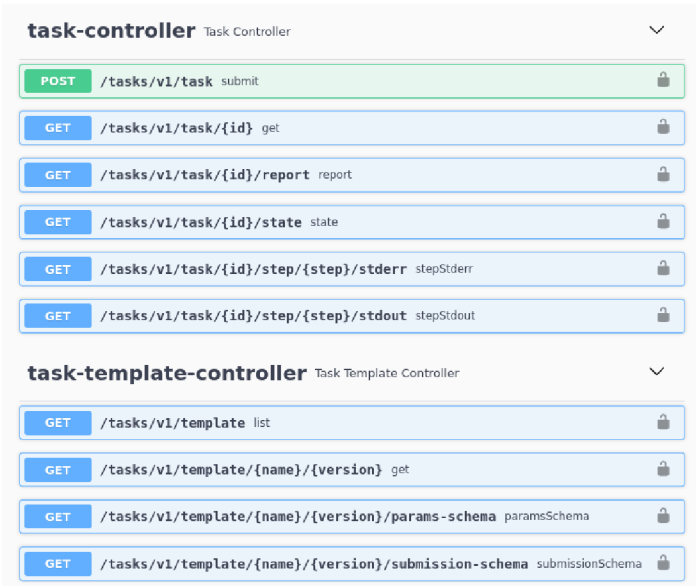

Figure 4.3: Available endpoints related to tasks execution. Website generated by integrated Swagger-ui package.

Once the task is completed, the user may review the process through endpoints designed for report browsing. The output files are stored in session-specific storage, and may be accessed using the storage API.

To extend LLWS with a new task, all that is needed is to add the three aforementioned definition JSON files to the Java project, and to a Python script to the Python library, which defines what commands to run with what parameters, and what outputs should be returned. Additionally, a Docker image with the Python library and all necessary additional software will be created. Endpoints, and therefore, clients, do not have to implement any changes as the communication protocol is not altered.

A sample code snippet that illustrates the process of executing a task from the client perspective may appear as follows:

```
Session session = Llws.sessionClient().create();

var client = Llws.tasksClient().vinaClient();
VinaTask task = client.newTask()
        .session(session.getId()
        .outputPath("vina")
        .params()
        .receptor(uploadedReceptor)
        .ligand(uploadedLigand)
        .grid()
        .centerX(10)
        .centerY(10)
        .centerZ(10)
        .end()
        .end()
        .submit();

VinaResults result = task.monitor().get();
result.getPoses().save(Paths.get("/poses.pdb"));
```

In this example, a new virtual sitting is created using a SessionClient class. In this sitting, a task running Autodock Vina calculation is submitted along with all necessary parameters using the TasksClient class. Each task object, representing a submitted task, then may auto-

matically await completion using the monitor() method. Last line of code then downloads output file to local storage.

4.2 Facades implementation

Facades allow the users to run jobs and submit requests to third-party APIs. Facades are implemented exclusively at the Java application level and have their own set of endpoints, where users can specify the parameters for running a given facade. Currently, there are three facades available: Matomo, PDB downloader, and HOPE API. Each facade calls different services and is implemented separately. However, the general logic is similar.

When a facade is submitted via an endpoint, it is registered in the database and assigned an ID, facade type, creation time, and status. All input parameters are stored in a locally stored JSON file, which allows for potential resubmission in case of an error. If the application crashes, any unfinished facade can be restarted and resubmitted using the values from this file. The results are stored in a raw text file, allowing for any type of return value from the facade.

After submitting a facade, the user is given a UUID, which can be used to monitor the status of the facade and retrieve its results. The endpoint for monitoring the existing facades is shared for all facade types.

The communication with third-party API is handled in a separate thread that exists until the facade is completely finished. The exact process of communication varies depending on the exact implementation of each API.

The fetching of a PDB file encompasses one of two scenarios. Either the PDB can be directly stored in the storage of a session provided by the user, or it can be acquired directly as the output of the facade execution. In either case, only a single request needs to be made to the PDB database[27][28]. The asynchronous nature of facades primarily serves to hide potential outages of the remote endpoints and allow for retrying communication in longer intervals.

Matomo facades communicates with Matomo[29] endpoints to obtain the number of visitors for a particular site. This is a feature that all tools in the Loschmidt Laboratories provide on the home page.

This facade only relays the API call from LLWS to Matomo. Although asynchronous workflow complicates the access to the data, it serves as a filter to potential technical issues.

HOPE facade is the most complicated one currently implemented. As HOPE API is also asynchronous, each run of a facade must submit a job, await its completion and then download the results. Each of these steps allow for a potential communication failure which may extend the runtime significantly. Here, the potential need of having to unexpectedly recover the state after an application crash is the most obvious and pressing. The solution to store values in files outside the application, has been put in place especially to handle more complicated facades. As similar facades are likely to be added in the near future, this approach ensures that the system can recover from any crashes and continue to provide reliable service to users.

The HOPE API is also noteworthy due to the API requiring the addition of a time-limited cookie, which is only generated after visiting the main website. Therefore, successfully submitting an API request necessitates programmatically accessing the website, extracting the token from the generated cookie, and incorporating it into the API request. In LLWS, communication with the HOPE API is accomplished through the use of the Retrofit library. The loading of the cookie is facilitated through Retrofit interceptors, which add the requisite token just before submission. By integrating this functionality into LLWS, it eliminates the need to repeatedly implement this logic in each tool within the Loschmidt Laboratories.

4.3 Parsers implementation

Parsers and tasks share many similarities. The most notable difference is in the API and typical workflow. Unlike tasks, parsers are not meant to perform any commutations. They are intended solely to read existing files and provide their contents in JSON format. Because of this, parsers run extremely fast and it was deemed needlessly complicated and too cumbersome to implement asynchronous API for an operation that runs within seconds. As a result, there is only a single endpoint connected to execution of parsers. Using this endpoint a user

may submit the request to parse a file and the resulting JSON will be returned in response.

Similarly to tasks, parsers are identified by name and version number. Each parser type is defined by two JSON files in the source code. The first defines input parameters schema, defining what information the parser needs from the user to be able to run. Second file holds the result schema, which is used to validate the output produced by the parser. Both schemas are also available from a separate endpoint, allowing users, who do not rely on LLWS client library, to obtain the structure of a parser output before execution.

To submit a new request, the user must provide the name of the parser to run, its version and parameters for execution. While a session ID is not strictly required, the files to be parsed are always downloaded from the S3 storage. Unless a static file is to be parsed, a session ID is also needed. They may also choose to upload the resulting JSON to their session storage. In such case, a session ID and path where to upload the file are needed.

After receiving a request and the parameters are validated, the Java application runs the appropriate parser. Parsers are written in Python as a part of the LLWS Python package. The parser itself downloads the file to be parsed, reads it, generates the JSON locally and optionally uploads the result back to S3 storage. The Java application then reads the local copy of the file and returns it as a response to the submit request.

There is no database or any other way to recover the state of a parser during an application restart or after it is finished. As parsers run very fast, it is unlikely the application would crash while the parser is running. If such unlikely event happens, the connection to user would close without any answer and the parser run would be lost. Then user is expected to resubmit the request, which will rerun the parser from scratch.

5 Current deployment and Outlook

The application, at its current form, is fully functional and incorporates all the features established as the project goals at the beginning of the development. The application supports user authentication with variable privileges. It can execute the bioinformatics tools and parse the output files of these tools with full abstraction over the implementation from the client perspective. The application also includes the communication with third-party APIs. All this is implemented with retrying policies in place and with full capacity to recover the state after a restart, where applicable. The versioning system of all endpoints guarantees backwards compatibility with client applications, which are using the implemented client library.

The application is ready to be integrated into most applications at the Loschmidt Laboratories.

LLWS has already been deployed in production, where it is being used by the PredictSNP Onco server, which was released for free public use in December 2022. PredictSNP Onco fully relies on LLWS as an execution platform. Its development and deployment allowed to test the fundamental principles of the architecture and provided useful insights into how to improve the usability by client applications, as well as new ideas for additional functionalities. So far, no major issues related to the application have been found.

Most of the tasks currently implemented are related to the PredictSNP Onco workflow, as it was the first tool built on top of this application. There are currently 22 available tasks, 6 parsers, and 3 facades. It is planned to migrate all the older tools to this platform in the near future, and efforts are already underway to integrate LLWS into the other tools. Most notably, FireProt Web, which is being worked on to finish and finalize the code for the new version 2.0.

The most basic changes will involve the addition of new tasks, facades and parsers that will be required by other tools in the Loschmidt Laboratories. The migration and reimplementation of other software may potentially reveal shortcomings or possibly suggest possible improvements.

In the long term, once LLWS is integrated into most or all applications, it will make it easier to develop new tools and maintain existing

ones as changes will be only needed in one place. Furthermore, since LLWS is capable of computing in multiple different environments, it also allows gradual transition from relatively old technology of PBS to more modern alternatives like Kubernetes or SLURM. This can be achieved by moving one task at a time, with no need to make drastic changes and large projects for migration from PBS.

LLWS was intended to be as cloud-native as possible, without impeding the functionality. Eventually, it will be possible to entirely abandon the current infrastructure of virtual machines running docker containers in favour of the Kubernetes cluster. Whereas this task would require enormous efforts within previous architecture. The level of abstraction that LLWS provides will without a doubt simplify this process and allow for changes to be less radical and more manageable in a small team of people.

It was also suggested that human users could be allowed to run calculations on LLWS using their own scripts. This would require creating a Python client, as Java is not a suitable programming language for scripting purposes. LLWS was initially intended to communicate with other applications, so an interactive bash shell on HPC node may prove easier to use than writing a script for LLWS. Nevertheless, it is an idea worth experimenting with, as there may be other potential use cases that will be found in the future.

A An appendix

The information system contains the following appendices in electronic form:

- The source code of LLWS application in Java
- The execution Python library labelled as llws-runners

Bibliography

1. JOHN WILSON, Tim Hunt. *Molecular biology of the cell*. 6th ed. New York: Garland Science, 2014. ISBN 978-0-8153-4453-7.
2. WHITFORD, David. *Proteins: Structure and Function*. Germany: Wiley, 2013. ISBN 9781118685723.
3. WOODLEY, John M. Protein engineering of enzymes for process applications. *Current Opinion in Chemical Biology*. 2013, vol. 17, no. 2, pp. 310–316. ISSN 1367-5931. Available from DOI: `https://doi.org/10.1016/j.cbpa.2013.03.017`. Bioinorganic Chemistry • Biocatalysis and Biotransformation.
4. DALBY, Paul A. Optimising enzyme function by directed evolution. *Current Opinion in Structural Biology*. 2003, vol. 13, no. 4, pp. 500–505. ISSN 0959-440X. Available from DOI: `https://doi.org/10.1016/S0959-440X(03)00101-5`.
5. ARNOLD, Frances H; VOLKOV, Alexander A. Directed evolution of biocatalysts. *Current Opinion in Chemical Biology*. 1999, vol. 3, no. 1, pp. 54–59. ISSN 1367-5931. Available from DOI: `https://doi.org/10.1016/S1367-5931(99)80010-6`.
6. STEFAN LUTZ, Uwe Theo Bornscheuer. *Protein Engineering Handbook*. 1st ed. New Jersey: Wiley VCH, 2012. ISBN 978-3-527-66701-7.
7. DVORAK, Pavel. *Methods of Directed Evolution and Their Application for Engineering of Haloalkane Dehalogenases*. 2007. Available from DOI: `10.13140/2.1.3648.0006`. PhD thesis.
8. JIRI DAMBORSKY, Jan Brezovsky. Computational tools for designing and engineering enzymes. *Current Opinion in Chemical Biology*. 2014, vol. 19, pp. 8–16. ISSN 1367-5931. Available from DOI: `https://doi.org/10.1016/j.cbpa.2013.12.003`. Biocatalysis and biotransformation Bioinorganic chemistry.
9. BENDL, Jaroslav; STOURAC, Jan; SEBESTOVA, Eva; VAVRA, Ondrej; MUSIL, Milos; BREZOVSKY, Jan; DAMBORSKY, Jiri. HotSpot Wizard 2.0: automated design of site-specific mutations and smart libraries in protein engineering. *Nucleic Acids Res.* 2016, vol. 44, no. W1, W479–W487.

10. MUSIL, Milos; STOURAC, Jan; BENDL, Jaroslav; BREZOVSKY, Jan; PROKOP, Zbynek; ZENDULKA, Jaroslav; MARTINEK, Tomas; BEDNAR, David; DAMBORSKY, Jiri. FireProt: web server for automated design of thermostable proteins. *Nucleic Acids Res.* 2017, vol. 45, no. W1, W393–W399.

11. HON, Jiri; BORKO, Simeon; STOURAC, Jan; PROKOP, Zbynek; ZENDULKA, Jaroslav; BEDNAR, David; MARTINEK, Tomas; DAMBORSKY, Jiri. EnzymeMiner: automated mining of soluble enzymes with diverse structures, catalytic properties and stabilities. *Nucleic Acids Res.* 2020, vol. 48, no. W1, W104–W109.

12. STOURAC, Jan; VAVRA, Ondrej; KOKKONEN, Piia; FILIPOVIC, Jiri; PINTO, Gaspar; BREZOVSKY, Jan; DAMBORSKY, Jiri; BEDNAR, David. Caver Web 1.0: identification of tunnels and channels in proteins and analysis of ligand transport. *Nucleic Acids Res.* 2019, vol. 47, no. W1, W414–W422.

13. MUSIL, Milos; KHAN, Rayyan Tariq; BEIER, Andy; STOURAC, Jan; KONEGGER, Hannes; DAMBORSKY, Jiri; BEDNAR, David. FireProtASR: A web server for fully automated ancestral sequence reconstruction. *Brief. Bioinform.* 2021, vol. 22, no. 4.

14. *Loschmidt Laboratories* [`https://loschmidt.chemi.muni.cz/peg/`]. [N.d.]. Accessed: May 1, 2023.

15. ALTSCHUL, Stephen F.; GISH, Warren; MILLER, Webb; MYERS, Eugene W.; LIPMAN, David J. Basic local alignment search tool. *Journal of Molecular Biology*. 1990, vol. 215, no. 3, pp. 403–410. ISSN 0022-2836. Available from DOI: `https://doi.org/10.1016/S0022-2836(05)80360-2`.

16. TROTT, Oleg; OLSON, Arthur J. AutoDock Vina: improving the speed and accuracy of docking with a new scoring function, efficient optimization, and multithreading. *J. Comput. Chem.* 2010, vol. 31, no. 2, pp. 455–461.

17. WRATTEN, Laura; WILM, Andreas; GOKE, Jonathan. Reproducible, scalable, and shareable analysis pipelines with bioinformatics workflow managers. *Nat. Methods*. 2021, vol. 18, no. 10, pp. 1161–1168.

18. *cwl-WES* [https://github.com/elixir-cloud-aai/cwl-WES]. [N.d.]. Accessed: April 17, 2023.

19. *Steep* [https://steep-wms.github.io/]. [N.d.]. Accessed: April 17, 2023.

20. KRÄMER, Michel. Efficient scheduling of scientific workflow actions in the cloud based on required capabilities. In: *Communications in Computer and Information Science*. Cham: Springer International Publishing, 2021, pp. 32–55. Communications in computer and information science.

21. *Apptainer* [https://apptainer.org/]. [N.d.]. Accessed: April 19, 2023.

22. TANZU, VMware. *Spring* [https://spring.io/]. [N.d.]. Accessed: April 19, 2023.

23. GROUP, The PostgreSQL Global Development. *PostgreSQL* [https://www.postgresql.org/]. 1996. Accessed: April 19, 2023.

24. TANZU, VMware. *Retrofit* [https://square.github.io/retrofit/]. 2013. Accessed: April 20, 2023.

25. AWS, Amazon. *Amazon S3 REST API Introduction* [https://docs.aws.amazon.com/AmazonS3/latest/API/Welcome.html]. [N.d.]. Accessed: April 20, 2023.

26. *Kubernetes OpenShift Java Client* [https://github.com/fabric8io/kubernetes-client]. [N.d.]. Accessed: April 22, 2023.

27. BERMAN, H M; WESTBROOK, J; FENG, Z; GILLILAND, G; BHAT, T N; WEISSIG, H; SHINDYALOV, I N; BOURNE, P E. The Protein Data Bank. *Nucleic Acids Res.* 2000, vol. 28, no. 1, pp. 235–242.

28. *RCSB Protein Data Bank* [https://www.rcsb.org/]. 2000. Accessed: May 1, 2023.

29. *Matomo Tracking HTTP API* [https://developer.matomo.org/api-reference/tracking-api]. [N.d.]. Accessed: April 17, 2023.