Protein Isoelectric Point Database

EE4FYP Report

MEng Electronic Engineering
& Computer Science

Student: Chris Cummins
Supervisor: Ian Nabney
Moderator: Zuoyin Tang

Date: May 2014

Electronic Engineering,
School of Engineering and Applied Science,
Aston University

Protein Isoelectric Point Database

EE4FYP Report

Chris Cummins

Abstract

The isoelectric point or pI of a protein corresponds to the solution pH at which the net surface charge is zero. Since the earliest days of solution biochemistry, the pI has been recorded and reported, and literature reports of pI abound. The protein isoelectric point database (PIP-DB) has collected and collated this legacy data to provide an increasingly comprehensive database for comparison and benchmarking purposes. As part of a collaboration between Aston University's Computer Science and Life and Health Sciences departments, a web application has been developed to warehouse this database and provide public access to this important information.

A schema and file format (YAPS) has been designed to house protein isoelectric point datasets, with appropriate tooling to convert between PIP-DB and the YAPS format. A search engine and domain specific language has been designed which enables searching of PIP-DB by representing compound queries using tree structures in LISP. Support for protein sequence searching has been implemented using the NCBI BLAST+ search tools.

A human-centred approach to designing web applications has been adopted, with a heavy focus on interaction design and usability testing. The results of usability testing show numerous advantages in the interface design, such as a widget which dynamically indicates the number of results to be returned by a search engine query.

A public API for allowing programmatic communication with the search engine has been designed. The website makes extensive use of mobile code, implementing a thin presentation layer wrapper around the API. The use of mobile code in websites is discussed and a comparison is made with server-side rendering of HTML.

A unique emphasis has been placed on development of infrastructure, with several new tools being written for reuse in other projects. Among these is the novel application of Markov text generators for creating test payloads from confidential datasets, and a project management program (pipbot) which automates version control and build configurations.

A parallelised build system has been developed which provides homogeneous development and deployment configurations. Tests show how the use of shell-level parallelism reduces build times by a factor of 5. An implementation of checksum based cache invalidation and on-demand build systems using the Linux's inotify subsystem is described.

Acknowledgements

I would like to thank Ian Nabney for the excellent continued supervision and guidance, without which this project would not have been possible. Further thanks to Darren Flower for providing the dataset and for patiently enduring my lowly understanding of the natural sciences. I would like to acknowledge Kate Sugden and all of the academic staff at Aston University who I've had the pleasure of being taught by. Special thanks to Fraser Crofts, Ben Stone, Shahzad Mumtaz, Mahmood Jasmin, and Dan Clarke for volunteering their time to help with usability testing.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

The purpose of the project was to take a set of data collated by members of the Life and Health Sciences department and to categorise and store it online in an accessible form for other people to search. There are many existing databases categorising biological information at the molecular level, but none for the isoelectric point (pI). The isoelectric point is the acidity (pH) at which a molecule carries no net charge. Below the isoelectric point, proteins have a net positive charge, above it a negative charge. In the denatured state, the pI depends solely on a protein's amino acid composition.

Professor Ian Nabney proposed the project and supervised development. Dr Darren Flower of the Life and Health Sciences department acted as the principle owner of PIP-DB, providing a copy for use in the website. Additionally, he was responsible for helping to explain the required biochemistry theory, and acted as a potential user during design feedback sessions. The two primary deliverables provided by Professor Nabney in the project description were:

1. An updatable relational database warehousing the provided dataset.
2. A web-accessible GUI with searching and downloading functionality.

This meant that the aim of development would be to get a website hosted online on a public server, accessible via a suitable web domain address. The website should offer a service whereby users can search PIP-DB and download search results using a web browser. This simple and open-ended specification allowed for the majority of development effort to be focused upon innovating on the areas of database driven website development, and search engine usability.

The primary reason for choosing the project was that there is a real existing need for a product of this type within the molecular biology community, with potential value and use for future scientific research. Development of database driven websites was an area which I was unfamiliar with and had no prior experience in, but is a skill that I felt was important to master.

This is an interdisciplinary project that involves aspects of web application development, bioinformatics, data analysis, interaction design, and data mining. The project plan used the OpenUP development process, which places an emphasis on the early mitigation of risks. Two sets of milestones were defined to govern project development, covering the requirements of the design and implementation aspects of the project. The design milestones were given the labels D1 through D4 and the implementation milestones M1 through M3. Figure 1.1 shows the chronology of development.

Objectives

The following objectives were written during before the planning stage of the project in order to provide a very high level overview of the finished product, and to express personal goals and expectations for the project which can then be refined to produce specific and quantifiable requirements.

1. To build a free (as in freedom) web application for searching and viewing protein isoelectric points.
2. To produce a bioinformatics tool with real world value for future scientific research.
3. The application should provide intuitive but powerful searching facilities.
4. The application should provide a convenient means for a certified user to edit and upload additional data.
5. The application should present information in a usable and efficient form.
6. Users should be allowed to download generated results for offline use.
7. Adequate security precautions should be taken to minimise the risk of data being sabotaged or stolen.
8. The implementation should use a clean model view controller architecture.
9. Comprehensive test coverage of the API and common use cases should be automated.
10. The application should be scalable for much larger datasets.

Chapter 2 includes a description of the requirements derived from these objectives, and Chapter 6 evaluates the project with reference to the objectives, offering a measure of achievement for each. See the Overview section for a thorough description of the document structure.

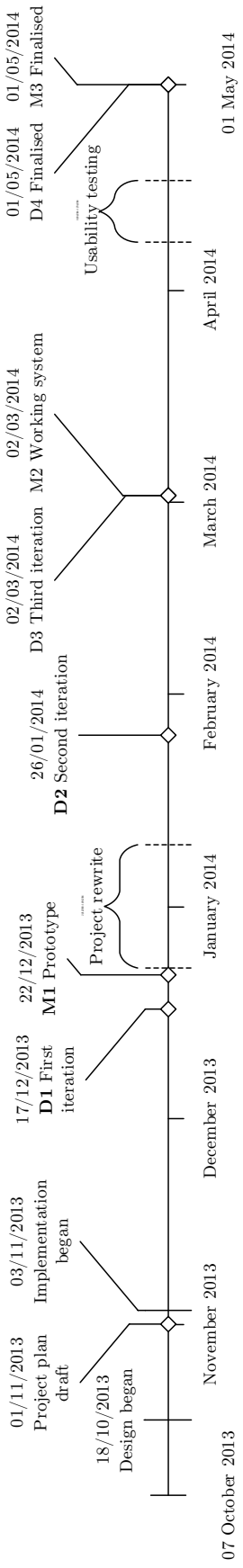


Figure 1.1: A timeline of project development, showing significant major project milestones.

Submission archive

The accompanying archive for this report contains a snapshot of the development repository. It was prepared automatically using the `mksubmission.sh` script written for this purpose. While every effort has been made to include relevant source code listings in this report, it is intended that a curious reader will investigate the contents of the archive of their own accord, as the large size of the source code means that only a small proportion can be reproduced in this report. Figure 1.2 identifies the key directories that may be of interest to the reader. The diagram shows a directory tree, where each directory has been annotated with numbers corresponding to the sections which discuss them within this report. The UNIX path delimiter "/" is used throughout this text, with absolute paths referring to the root directory of the archive.

This is a polylingual project including source code written in Clojure LISP, JavaScript, Less CSS, M4sh, Make, Python, and sh programming languages. The documentation is formatted in LaTeX, HTML and Markdown. A reasonably competent text editor is all that is required to view the source files, which can be identified by the file extensions: ac, am, bib, clj, fsa, in, js, json, less, md, py, sh, tex, xml, yml.

Note that while the submitted archive contains all project source codes, the third party BLAST+ binaries and compiler JARs have been removed in order to keep the size of the submission archive down.

Archive directory	**Sections in this document**
/	
├─ bin	*4.1*
├─ build	*4.1*
├─ Documentation	*3.1, 3.2*
│ ├─ design	*3.2*
│ ├─ evaluation	*6.1*
│ ├─ midterm	*3.1, 5.2*
│ ├─ plan	*5.1*
│ └─ report	
├─ extern	*5.5*
├─ resources	*4.1*
│ ├─ css	*4.1*
│ ├─ fonts	
│ ├─ img	
│ └─ js	*5.5*
├─ scripts	*4.3*
├─ src	*5.3, 5.4, 5.5*
├─ test	*4.2*
└─ tools	*4.3*
├─ csv2yaps	*5.4*
├─ png	*4.2*
├─ watchr	*4.1*
└─ yaps2fsa	*5.4*

Figure 1.2: Archive directory structure and report section cross-references.

Building the project

A complete copy of the project can be compiled on a GNU/Linux operating system by executing the command `./bin/build --all` from within the project root directory. This will invoke a script that will attempt to automatically resolve system dependencies and requirements. Failure to meet the system requirements will results in an informative error message being displayed. See Section 4.1 for further information.

Nomenclature

Acronyms and initialisms will be used where appropriate in place of full terms. When one is to be used, the first use of the term will include the full expanded name, followed by the acronym in parenthesis. The acronym will then be used from thereon in.

On the project name

Throughout this text it is necessary to carefully distinguish conversation about the biological dataset from the software and algorithms used to host it. To achieve this distinction, I will use the initialism PIP-DB to refer to the biological dataset assembled by members of the Life & Health Sciences department, and the *name* pip-db to refer to the software project that I developed to host this. In order to keep this distinction clear, the capitalisation will be kept consistent irrespective of the grammatical context.

On the use of UML

A subset of the Unified Modelling Language [1] is used a basis for many of the technical diagrams. One notable deviation from convention is the use of sequence diagrams to describe the behaviour of web services [2]. In such cases, the desired effect is to illustrate the behaviour of a system at a high level, not to provide a technically accurate description of communication.

Overview

The rest of the text is structured as follows: the project risk assessment is described in Chapter 2, with a brief overview of the risks identified during the project planning phase and a description of the mitigation strategies. A large body of text is then devoted to a description of the design and implementation of the final product, and is divided accordingly: information relevant to the development process and work flow is described in Chapter 3; the development of infrastructure and tooling is described in Chapter 4; and the product implementation in Chapter 5. Chapter 6 contains an evaluation of the software and the results of usability testing, and the conclusions and recommendations for further work can be found in Chapter 7. The appendices begin at page 48, followed by the bibliography and references at page 59. The purpose of the Requirements and Process chapters is to offer an insight into the development plan and workflow. A reader who is solely interested in the implementation and engineering of the project may skip ahead to Chapter 4.

Chapter 2

Risk Assessment

It is expected that 400 hours of work be put into a final year project. With such a large body of time dedicated to it, it is important to establish a clear direction for development, such that the project progress can be assessed at regular intervals along the project life cycle. The aim of progress assessments is to ensure that time is not wasted on work which does not positively contribute to the development of a finished product. In single-developer projects there is an even greater priority for regular progress assessments than in a large team project, as the single-developer is not accountable to anyone, leading to a greater chance of the project losing focus or suffering from second-system effect [3].

To reduce the chance of this, a set of project risks were identified during the planning stage (Table 2.1), and development was focused around the early mitigation of these risks.

For each risk, the probability of it occurring and the impact that it would have on project progress was assigned a numerical value between 0-5, allowing the risks to ordered in terms of severity using the geometric mean of these two values. High level risk mitigation strategies were constructed (Appendix A), and used as a starting point for creating the project life cycle plan, discussed in Section 3.

Risk	Description	Category	P	I
R1	Design is not intuitive	*Design*	2	3
R2	Project involves use of new technical skills	*Development*	5	5
R3	High Level of technical complexity	*Development*	5	3
R4	Complex deployment of production website	*Development*	5	4
R5	Project milestones not clearly defined	*Planning*	1	1
R6	System requirements not adequately identified	*Requirements*	2	5
R7	Change in project requirements during development	*Requirements*	1	5
R8	Changes in dataset format during development	*Resources*	2	5
R9	Unable to obtain required resources	*Resources*	1	1
R10	Users not committed to the project	*Users*	2	4
R11	Lack of cooperation from users	*Users*	1	4
R12	Users with negative attitudes toward the project	*Users*	1	2

Table 2.1: Project risks. The **P** and **I** columns assign numerical values to the each risk's probability and impact respectively, within the range 0-5.

Chapter 3

Process

The following chapter describes the processes used in the design and implementation of the project. Section 3.1 provides an explanation of the development process and the approach to implementation, and Section 3.2 describes the user-centred aspects of the design process.

3.1 Development process

A project with an individual developer requires a different approach to time management than a multi-developer project. The lack of other team members means that the development can afford to take a much more flexible and dynamic approach, allowing for a faster pace and lower cost of change in the development life cycle. Agile software processes focus on this fast pace of change by encouraging frequent communication with stakeholders and very short development iterations [4, 5].

Elements of the Agile manifesto [6] inspired decisions in the project development. For example, frequent meetings with Dr Flower were used to provide ongoing feedback of development progress, and meets the agile requirement for *customer collaboration over contract negotiation*. Additionally, the requirement for *working software over comprehensive documentation* was used to justify the early development of a working prototype which users could interact with, instead of lengthy requirements notifications with potential users before beginning development work. This resulted in useful design feedback during early stages of the project development, as it is more intuitive for users to provide feedback for a functional prototype then it is to discuss requirements in a more abstract manner without being able to interact with a product.

Not all of the Agile software philosophy was strictly adhered to. In particular, the emphasis on *responding to change over following a plan* was supplanted by the requirement for a formal project plan document to be created in the first term. Since the Agile philosophy doesn't provide a template or process for guiding development, so the iterative OpenUP development process was used to provide a time management framework, by dividing the development life cycle into discrete increments, each consisting of an inception, elaboration, construction, and evaluation phase [7]. OpenUP was chosen for the development process due to its entirely open source nature as part of the Eclipse Process Framework, and because it targets small teams and agile development by design [8]. Development was split over three iterations, with one covering the first term, and two in the second term. The end of each iteration's evaluation phase culminated in a design and implementation milestone pair.

3.1.1 Version Control

The git version control system was used to provide version control. A single monolithic git repository tracks revisions for all pip-db source codes and associated data. By using version control from the very project's inception, a fully accountable and transparent history of development has been recorded, with 2,420 revisions committed since 14 October 2013.

Git was chosen as the version control system due to its support for lightweight branching and distributed-by-design nature [9]. While the benefits of distributed version control are not entirely exploited for single-developer projects, the support for concurrent development of branches encourages experimentation and an agile approach to development.

3.1.2 Open Source

One of the key considerations of the project objectives (Chapter 1) was that the finished project should be freely available without commercial interest, and this extends to the source code and development process. The success of truly open models of development have been investigated in great detail [10–13], and a philosophy of "release early, release often" has been encouraged in open source communities as a technique for nurturing rapid and widespread user involvement from an early stage [14]. As a result, all of the program code, documentation and supporting files that have been created for pip-db have been released under the terms of the GNU General Public License v3. This is an open and permissive license that allows for commercial use, but it mandates that derivative works maintain the same license and must distribute the source code openly [15].

The combination of an open source license and the use of git version control meant that online repository hosting could be used to provide a public centre for development. For this, the GitHub website was used, which is the most popular online repository hosting site [16], and offers free hosting for open source projects [17].

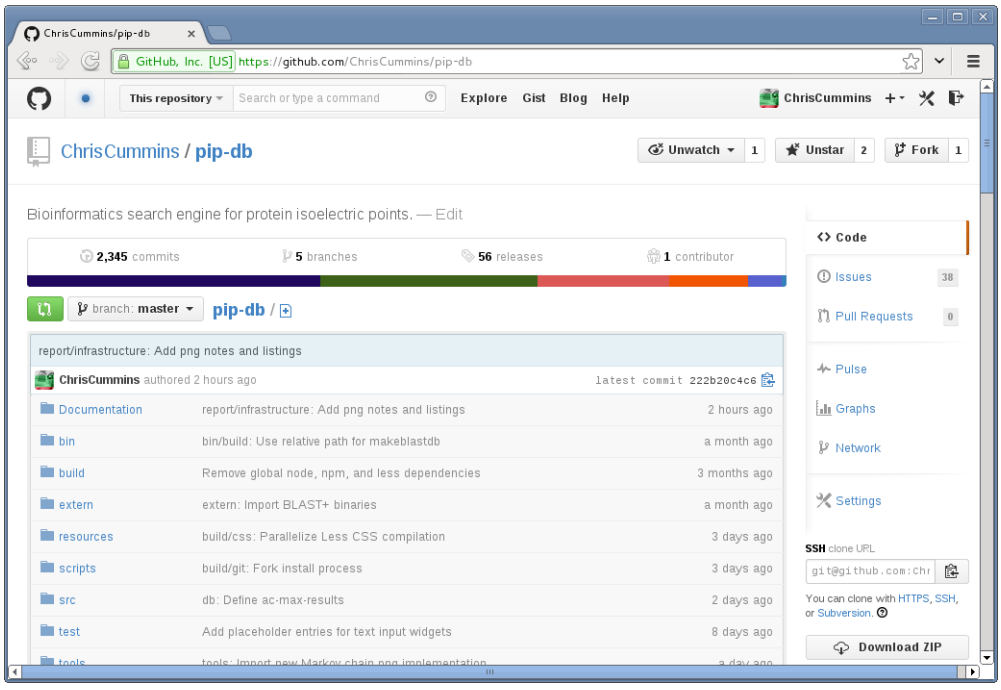


Figure 3.1: Screenshot of the GitHub repository for pip-db.

A note on dataset confidentiality

It is important to note that while pip-db is an open source project, the PIP-DB dataset as supplied by Dr Flower remains confidential at his request, and so has not been released for distribution.

3.1.3 Development workflow

In addition to hosting the project source code and revision history, GitHub provides many useful features intended for collaborative development efforts, including an issue tracker, milestones, and a Wiki. By combining these features with strict version control practises, it is possible to create a dynamic development environment which simultaneously encourages experimentation and rapid change while providing a full history of revisions and the ability to roll back and integrate new features when required.

The issue tracker provided by GitHub was used from the project inception. The purpose of the issue tracker is to document requests for changes. Whilst the revision control log provides a history of all of the changes which have been made, the issue tracker is used as a place to document changes which *should* be made, but not have not yet been completed. Each issue is assigned a unique identifier, and these identifiers can be used in revision messages to provide a cross reference between the revision control history and the known issues and bugs. In total, 350 issues have been opened, of which 39 remain open at the time of writing.

Issues can be categorised using labels (Table 3.1). Labels provide additional meta data regarding a type of issue, and each issue can be assigned multiple labels. The issue tracker can filter issues by labels, allowing for a quick visual overview of the issues particular types.

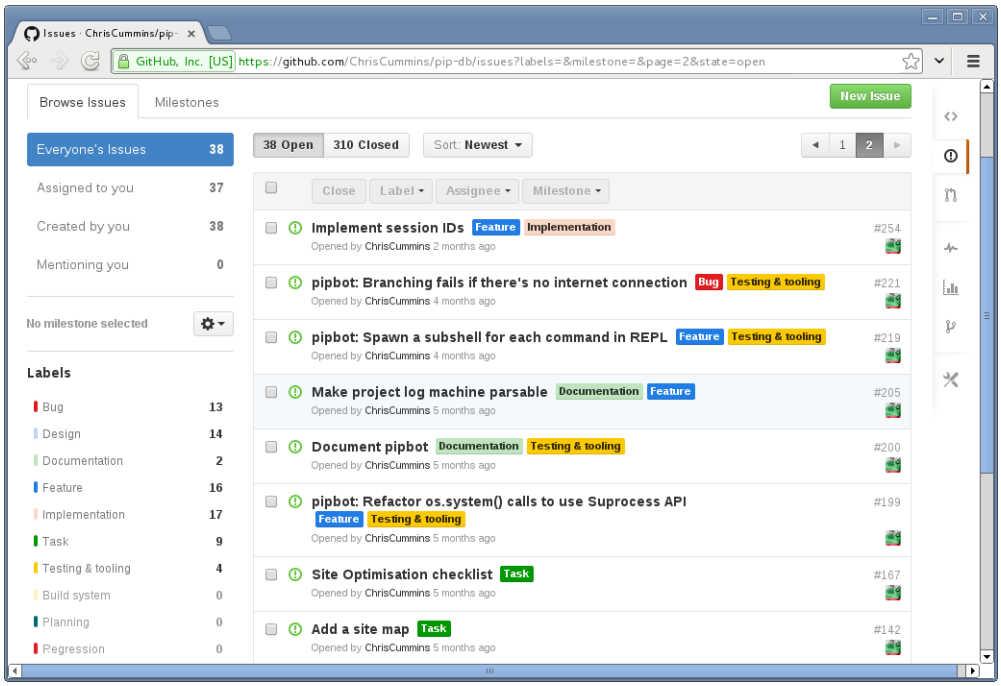


Figure 3.2: Screenshot of GitHub's issue tracker for pip-db.

Label	Description
Bug	Crash reports, stack traces, and software failures.
Design	Issues relating to the user interface design.
Documentation	Documentation tasks.
Feature	Web service feature addition requests.
Implementation	Issues relating to the web server implementation.
Task	Feature addition requests.
Testing & tooling	Issues relating to infrastructure.
Build system	Bugs and issues with the build system.
Planning	Issues relating to TP1 project planning.
Regression	Issues which have arisen as a result of changes, not additions.
Version control	Git and GitHub issues and feature requests.
Wontfix	Used to indicate issues which have been closed without being fixed.

Table 3.1: The labels used for categorising issues, and their corresponding meanings.

In addition to assigning labels to issues, GitHub also supports the creation of milestones with set dates. Issues can be assigned to milestones, and the number of open and closed issues per milestone can be shown, providing an indication of the progress towards a particular milestones (Figure 3.3).

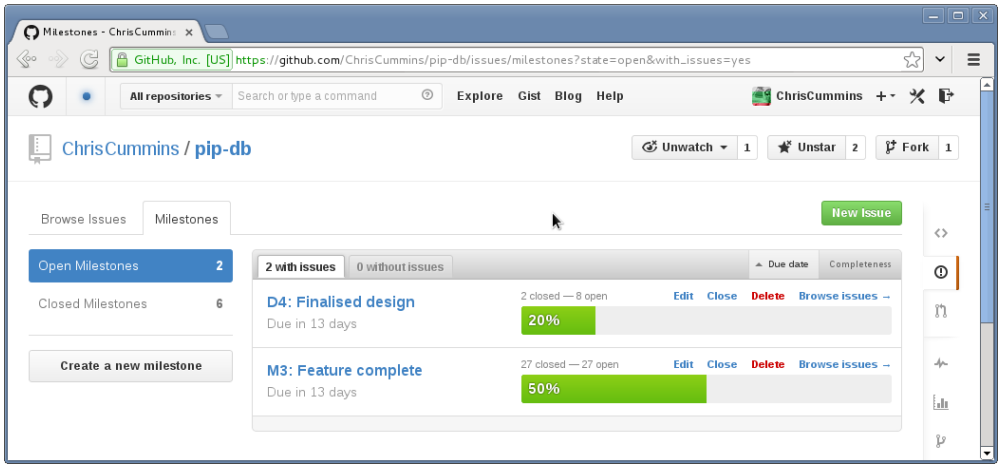


Figure 3.3: Screenshot of GitHub's milestones overview for pip-db.

3.1.4 Branching model

A branching model was designed for this project in order to provide a consistent branching and release strategy to use over the course of development. Since the revision control must track a huge number of changes (over 2000) over the course of several months, it is important that the revision history be as clear as possible, and that branches are used intelligently to provide additional information about the project development, not to obscure past work.

The branch model that I designed was inspired by Driessen's *Successful Git branching model* [18], with a number of changes to adapt it specifically for this project. The core of Driessen's model is two permanent branches which track the current development head, and the latest stable release. Developers work on the development branch, and update the stable branch at release time. Transient auxiliary branches are used as staging areas

for new features and releases. The hotfix branch support was removed from Driesson's model, since the software developed is only proof and concept and so does not need to support regression patching.

Driesson	Cummins	Purpose
`master`	`stable`	The latest stable release.
`develop`	`master`	The current development head.
`release/:name`	`release/:version`	Release candidate staging areas.
`feature/:name`	`wip/:id`	Feature addition branches (cross referenced with issue tracker using issue IDs).
`hotfix/:name`		Hotfix development branches.

Table 3.2: A comparison of branch names with Driesson's development model.

Importantly, the name scheme for feature branches was changed so that it matched the GitHub issue IDs. This meant that work on a feature should only begin when it has an open issue assigned to it, enforcing the use of the issue tracker. This novel integration of issue tracker and version control system means that for every change made in the project, it is possible to trace back not only the revision which introduced the change, but also the issue or feature request which demanded it. This means that every change is justified with a reason *why* the change was made, not just the description of *how* the change was made which is provided by the revision history.

3.1.5 Auxiliary documentation

To ensure a consistent use of version control, a checklist was created for each common activity (submitting a patch, creating a release, starting work on a new feature). Additional files document the high-level workflow and approach to release management. Relevant documentation in the submission archive includes:

```
Documentation/ReleaseChecklist.html
Documentation/SubmitChecklist.html
Documentation/SubmittingPatches.html
Documentation/Workflow.html
Documentation/VersionNumbering.html
```

An engineer's log was used to keep daily notes of all development activity, minutes from stakeholder meetings, and other tertiary information that is missing from the revision history and issue tracker. A HTML render of the log can be found in the submission archive `Documentation/Log.html`. The project log was written in Markdown format for quick typesetting, and used a consistent date format to separate individual entries which meant it was possible to parse the log using a simple Python script (`scripts/parselog.py`) for export into different formats and analysis (Table 3.3).

Number of log entries	127
Total word count	21,329
Average entry word count	167
Shortest entry word count	6
Longest entry word count	2,008

Table 3.3: Project log details.

3.2 Design process

The complement of the development processes are the design processes, which dictate the approach to user interface design. Both the project objectives and risk mitigation strategies emphasise the importance of creating an intuitive user interface for the search engine, making the decisions in design processes important to the success of the project.

3.2.1 Human-centred design

The software industry is undoing a renaissance in its attitude towards usability. In the post text-interface age, the graphical user interface has become the focus of many interesting shifts in design methodology. Human-centred design is a school of interaction design which edifies *usability* as the primary goal of all design [19], and encourages designers to take a measured approach to interaction design, since "the joy of an early release lasts but a short time. The bitterness of an unusable system lasts for years".

Uptake of human-centred design principles has been slower in the field of bioinformatics, which may be due to the common belief that usability is only of secondary importance to functionality, instead of being a core component of it. The disadvantage of this is that a number of popular existing bioinformatics tools have relatively poor usability, although there is evidence of attempts at innovation in some of the more popular tools [20, 21]. There have been efforts made to introduce human-centred design into bioinformatics, with studies showing that "although users believe that the bioinformatics community is providing accurate and valuable data, they often find the interfaces to these resources tricky to use and navigate" [22]. The importance of good usability as a time saving mechanism has been justified as "usability 'barriers' can pose significant obstacles to a satisfactory user experience and force researchers to spend unnecessary time and effort to complete their tasks" [23].

Over the course of the project development, a significant emphasis was placed on taking a human-centred approach design, not only because of the time saving benefits of a well designed tool, but also as a necessary coping mechanism for the difference in usability expectations between the computer science and biochemistry communities. Further risk mitigation is provided by extensive analysis of existing bioinformatics tools in the project planning stage[1].

3.2.2 Prototype development

Rapid prototyping played a key role in being able to achieve the goal of human-centred design, allowing for immediate and visual feedback from stakeholders and potential users. The project used two types of prototyping: low fidelity and high fidelity.

Low fidelity prototyping

Low fidelity prototyping involves the rapid generation of non-functional "paper prototypes" which are designed to give a rough impression of how the finished product will look, without specifying an implementation design. Low fidelity prototyping was achieved in this project using the Balsamiq[2] program to generate mockups and wireframes of the

[1] See the project plan for a critical analysis of existing bioinformatics search engines. A copy of the project plan can be found in the submission archive `Documentation/ProjectPlan.pdf`.

[2] Balsamiq. Rapid, effective and fun wireframing software. `http://balsamiq.com/`

pages of the website (Figure 3.4). These mockups were discussed with Dr Flower to verify that they were broadly satisfactory, and then used as the basis for the M1 implementation. After the M1 implementation was complete, the mockups were updated and refined for the second D2 design milestone (Figure 3.5).

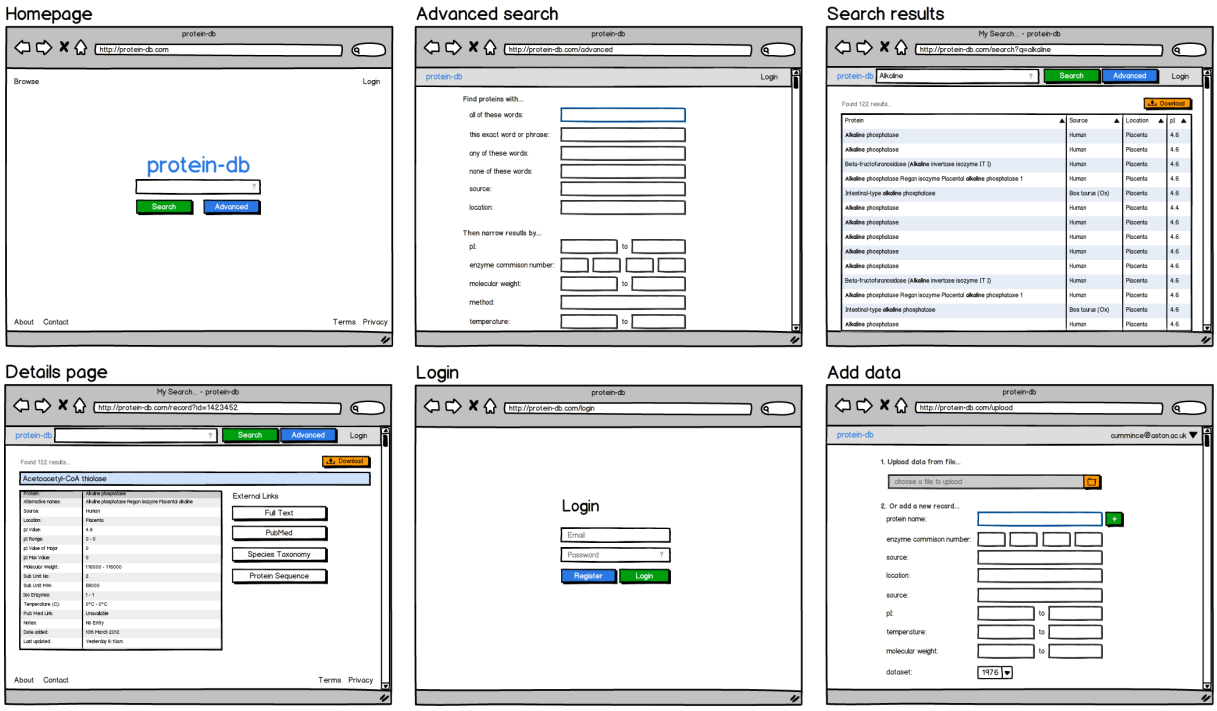


Figure 3.4: D1 design mockups for site pages.

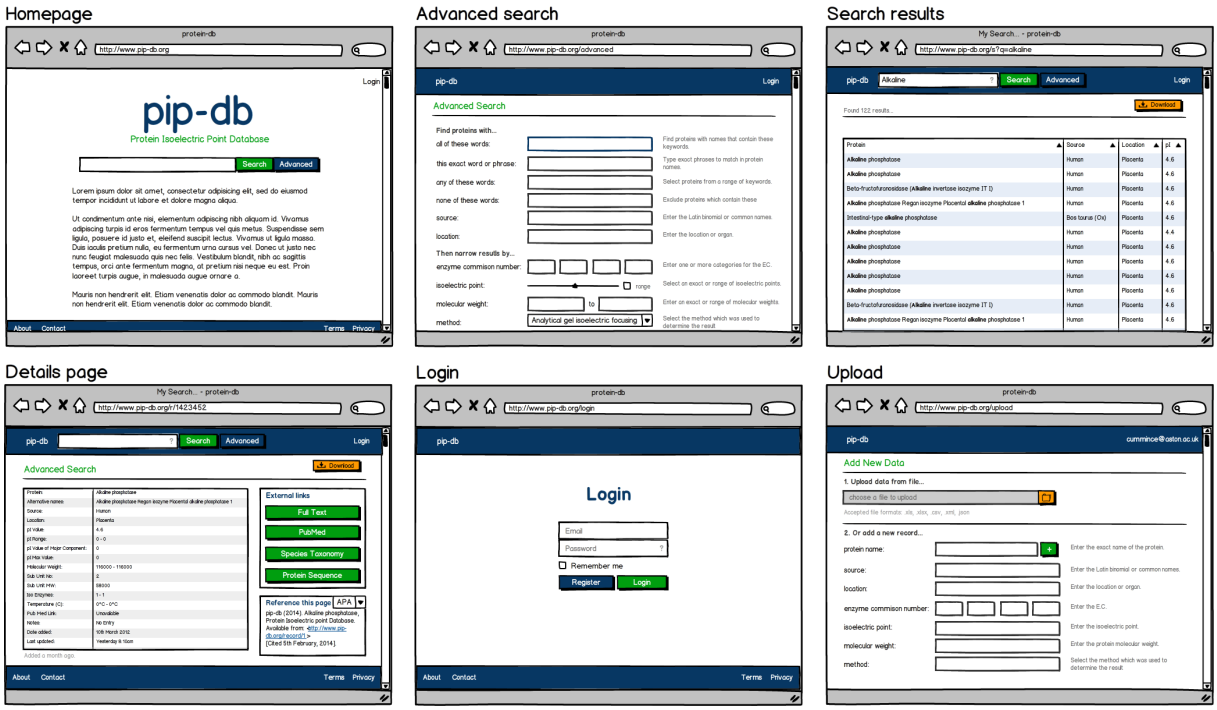


Figure 3.5: D2 design mockups for site pages.

In addition to providing a guideline for the page aesthetics, the mockups were used to give a rough outline of the interaction design. A list of common tasks was created and for each, a sequence of mockups was generated which show the steps that the user would have to take to achieve them. Figure 3.6 shows the interaction design for a simple name search using the D1 design. Successive design iterations refined these interaction designs further, and covered a wider variety of use cases, such as exception handling and error cases.

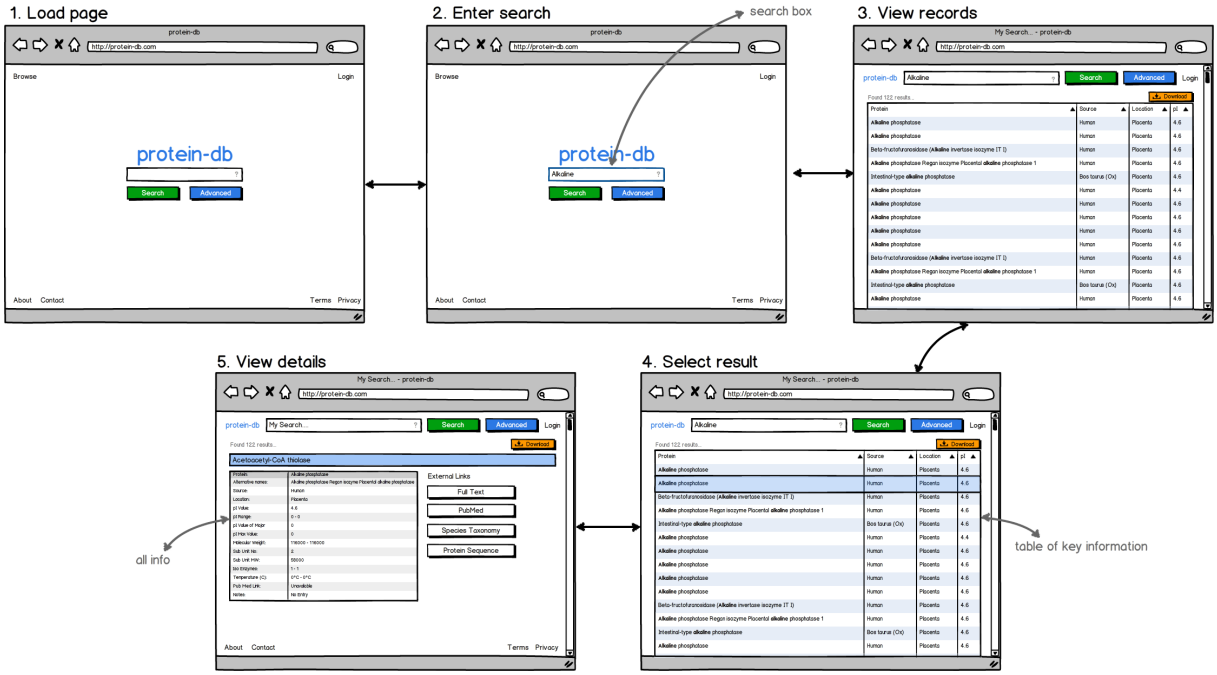


Figure 3.6: D1 interaction design for a simple use case.

High fidelity prototyping

In contrast to low fidelity prototypes, the purpose of a high fidelity prototype is to provide a way for potential users and stakeholders to interact with a tangible simulation of the final product [24]. This depth first prototyping requires a more technical approach which maps the high level design ideas of the low fidelity prototype to a functional back-end to provide dynamic behaviour.

In order to mitigate the risk of creating a tool which is not intuitive, an initial prototype implementation was required for the M1 milestone, at the end of the first term. This meant that the second term development could be focused upon refined and innovation of the user interface. There was a tight time schedule for development of this first prototype, so an opportunistic approach to programming was adopted, making maximum use of existing tools and supporting software where available [25]. The implementation of the M1 prototype is discussed in Section 5.1. See Appendix B for a comparison of the D1 low fidelity mockups with the M1 prototype implementation.

Chapter 4

Infrastructure

The purpose of infrastructure is to develop a set of tools to assist in the creation of a product by automating specific sets of actions or behaviour. In software projects, these tools broadly fall into one of three categories: build automation, testing automation, and task automation. In all cases, their primary purposes is to save developer time and reduce the risk of human error by automating repeatable tasks. The development of tooling was assigned a high priority throughout the project due to its mitigating effects on the risks of technical complexity and deployment complexity. Additionally, it is often possible to repurpose tools for multiple projects, which helps satisfy the project objective of producing something of real world value for future work.

4.1 Build automation

The building and compilation stage of pip-db requires the execution of many steps, the most notable of which are:

- **Clojure compilation** The web server is written in Clojure LISP, which is a compiled language that translate to Java bytecode for execution on the Java Virtual Machine (see Section 5.2).
- **JavaScript minification** Client-side scripts are pre-processed using a minifier, which is program that reduces the size of a source code by removing all unnecessary characters, and further reduces the size of the source code through semantic evaluation of the code in order to produce shorter variable names and generate compacted source code. The purpose of generating a compacted file is to reduce the bandwidth required to transmit it to the client, resulting in faster page loading times.
- **Stylesheet pre-processing** Client-side stylesheets are written in Less CSS, which is a language which compiles to CSS and offers extensions to the CSS specification, such as variables and control flow. Additionally, compiled CSS stylesheets are minified to reduce their size.
- **Documentation rendering** User documentation is compiled from intermediate forms (LaTeX, Markdown) into publishable formats (pdf, HTML).
- **Binary installation** Executable scripts and applications and must be installed into a appropriate directory in the system path, for example `/usr/local/bin`.

4.1.1 Build system

In order to avoid having to manually execute every step of compilation by hand, a build system is used to automate the process. This project uses the GNU Build System[1] (also known as Autotools) for the unorthodox purpose of compiling website resources and source code. Autotools is a component of the GNU toolchain that was designed for creating portable Makefiles by offering three levels of increasing abstraction for the build system:

1. **autoconf** The autoconf tool generates a configure script by scanning the build environment and generating scripts from input sources.
2. **configure** The configure script is responsible for detecting platform-specific variables and configuring the build, allowing for user configurations. Makefiles are generated at the end of configuration.
3. **make** Invoking the make program will build the project using the Makefiles, which contain a set of targets for performing tasks such as compilation, deletion of generated files, and exporting releases.

In addition to automating each of the compilation tasks mentioned above, the autotooled build system offers the following additional behaviours:

- **Cross-platform compatibility** Differences between platform-specific implementation details are handled by using the substitution of generics. This means that the build system will adapt to different environments and configurations.
- **Dependency resolution** A dependency tree models the relation of compiled sources, so that compilation can be ordered correctly to satisfy any dependencies. Additionally, third party packages are downloaded and installed as required.
- **User configurations** Run time flags for the configure script allow for partial compilation, enabling and disabling features, and setting the optimisation level for compilation.

Figure 4.1 shows how the build system prepares the HTML, CSS, and JS source files at build time. Autotools was designed for creating portable Makefiles for C software packages on UNIX systems. The application of Autotools for building websites is unusual, but the fine grained control and abstraction that is provided allows for a powerful and adaptive build system, with unique advantages over existing systems. Two such advantages are the ability to perform checksum based cache busting, and shell-level parallelisation of the build process.

[1] automake `http://www.gnu.org/software/automake/manual/automake.html`

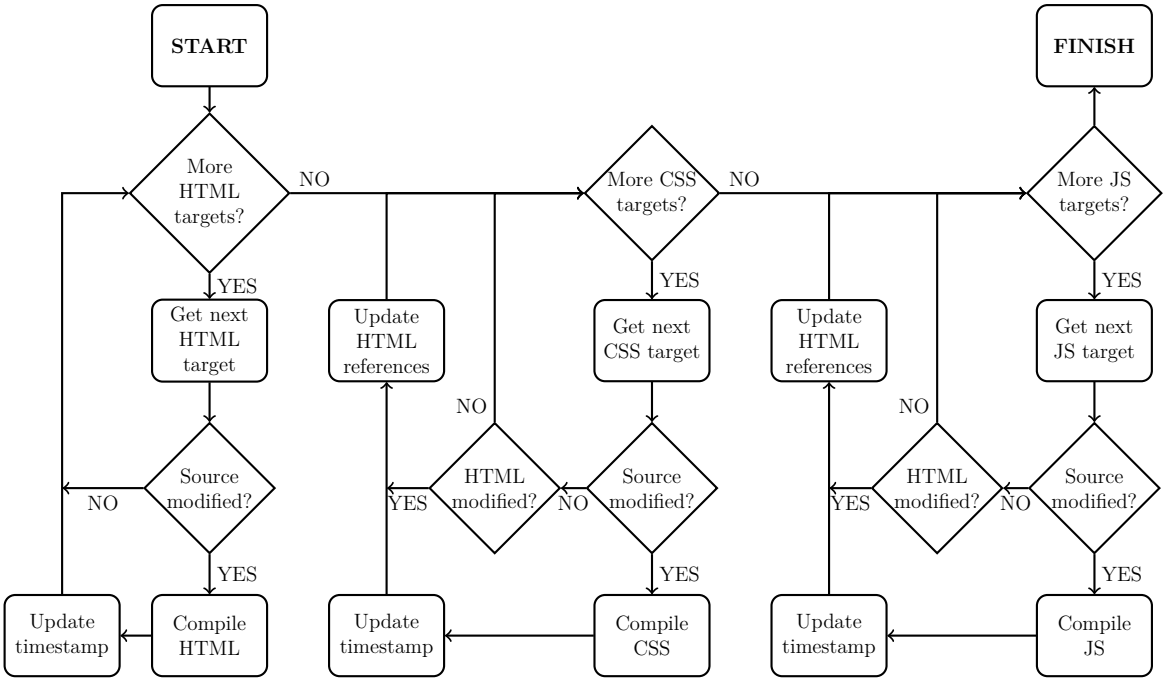


Figure 4.1: Flowchart of build system HTML, CSS, and JS subsystem.

Checksum based cache invalidation

When a web server transmits a file to a client, it is possible to indicate in the HTTP response headers that the client should cache a copy of the file for future use. That way, future attempts to retrieve the file allow the client browser to simply retrieve the local copy from the cache, removing the need for a HTTP round trip. This greatly reduces the load time on repeat requests for the same file, and so is commonly used for caching assets that are common to every page on a website, such as images, stylesheets, and scripts. The disadvantage of enabling this kind of caching is that if the contents of the file is modified after it has been cached by a client, then future requests for that file will retrieve the out of date cached copy. To prevent this, it is necessary to adopt a cache invalidation technique to ensure that files are not retrieved from the cache when they have been modified on the server.

One technique for invalidating cached files on modification is to use a cache naming scheme [26]. This means that assets are given version numbers, and those version numbers are incremented every time a modification is made. For example, if a HTML page included a script `main.js`, the file would be given the name `main-v1.js`, that would be cached by the client. If `main.js` was modified, it would be renamed to `main-v2.js`, and all references to it would be updated to reflect this new name. At this point, the `main-v1.js` file would remain in the client's cache, but the HTML page would now point to `main-v2.js`, causing the client to fetch this new modified file. This process can be repeated indefinitely, so long as unique file names are always assigned.

This technique for file name cache invalidation (known as *cache busting*) is easy to incorporate into a build system. Every time the build system executes, it increments the version counters for each file, and updates any references to that file to point to the new version. However, this causes unnecessary duplication of files, since there is no means for detecting whether a file has actually been *modified* when blindly increasing the version number.

```
def compile_javascript_source(file):
    if content_hashing = enabled:
        checksum = md5sum(file)
        target_name = basename(file) + substring(checksum, 8) + '.js'
    else:
        target_name = filename(file)

    if not file_exists(target):
        if minify_javascript = enabled:
            target = minify(file)
        else:
            target = copy(file)

        if content_hashing = enabled:
            for h in html_source_list:
                replace references to 'file' with 'target in h
```

Listing 4.1: Pseudocode listing for compiling a JavaScript source, with optional content hashing and minification.

The pip-db build system uses a more pragmatic approach, whereby for each file, its checksum is computed using the md5sum algorithm. A concatenated version of this checksum is then appended to the filename, and references to the file are updated to point to this new file name, incorporating the checksum. The benefit of calculating file checksums instead of using incrementing version numbers is that the checksum is dependent on a file's *contents*, and so will only change when the file has been modified. This succinctly solves the problem of cache invalidation only on file modifications, as unmodified files will have unchanged checksums. See Listing 4.1 for a pseudocode implementation of this cache busting algorithm.

Build system parallelisation

Autotools uses a recursive build system in which each directory is visited in turn, and all of the required compilation steps are executed before leaving and moving on to the next directory. This works well for the common use case of C programs, in which it may be necessary to generate all object files before invoking a linker. However, for the purposes of pip-db, this causes needless serialisation of the build system since there are very few dependencies between different files. For example, the compiled web server does not depend on the stylesheets, which in turn do not depend on the client side scripts. Despite this, each item is compiled sequentially, with the build system only beginning on the next item once the previous item has completed.

By considering the build process as a loop in which each iteration causes a new file to be compiled, it is evident that the loop could be successfully parallelised such that each iteration is assigned a separate thread, providing that there are no data dependencies between successive iterations. Using this technique, several successive parallelisation optimisations were made to the build system so that as much of the compilation work is performed in parallel as possible.

An experimental environment was set up on a development machine in which compilation was performed and timed 5 times, before applying a parallelisation optimisation and repeating the experiment. Figure 4.2 shows the results of the experiment. In each case, the optimisation was implemented by spawning the compilation task in a forked subshell, as opposed to executing the compilation process from the main build thread. The first test

performed was to time the serial build system, which executed in an average time of 39 seconds. The subsequent optimisation passes parallelised the following tasks: JavaScript minification, API documentation generation, CSS minification, image compression, Less CSS pre-processing, and tools compilation.

As can be seen, the greatest reductions in times were achieved by parallelising the CPU intensive JavaScript minification and API documentation generation. Compilation tasks which are not CPU bound such as image compression achieved a much lower reduction in compilation time through parallelisation, and in fact generated a very slight increase in the execution time. This is due to the cost of spawning a new subshell which is incurred by using shell-level parallelisation techniques. The build system parallelisation optimisations reduced execution time by a factor of 5. Further execution time reductions could be achieved by parallelising the recursive directory traversal of Autotools, but this would require significant alterations to the core automake package which are beyond the scope of this project.

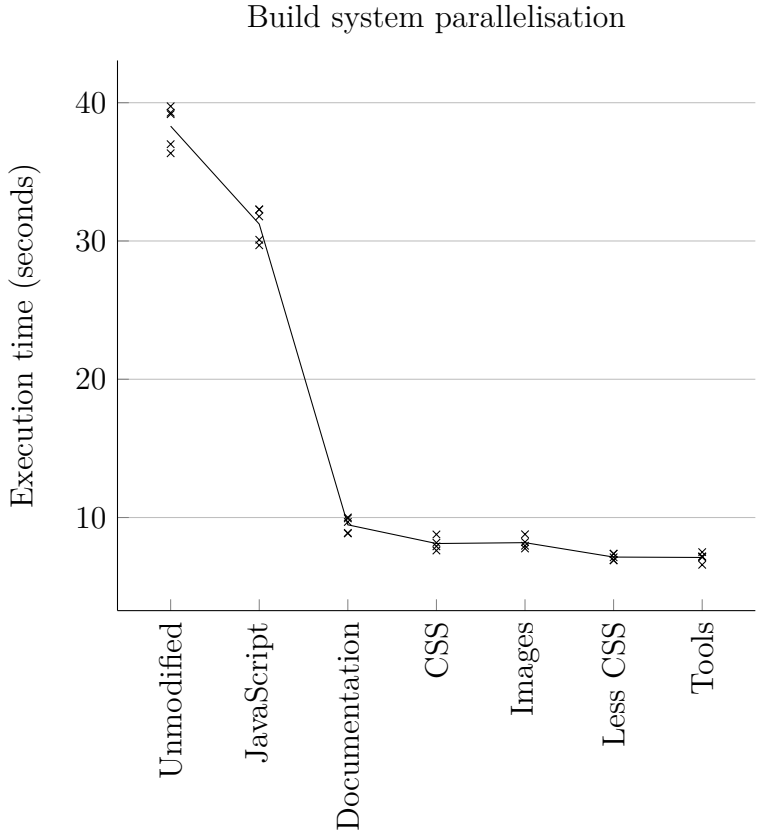


Figure 4.2: Graph of build system execution times after each optimisation iteration. Each value is sequential, with the leftmost control value being the execution time of the unmodified serial build system. The tests were invoked with the command `time (./autogen.sh && ./configure && make)`.

4.1.2 On-demand compilation using inotify subsystem

A common desire amongst software engineers is to reduce the feedback loop when programming between making a modification to a source file, recompiling the sources, and executing the compiled program. The Autotooled build system described above provided a means for performing automated compilation of the project sources, but it must be manually invoked by the user, and it does not run any programs once completed. In order to reduce the feedback cycle for pip-db, a script was written which utilises the real time event notifications provided by the Linux inotify subsystem [27, 28], in order to invoke the build system whenever a dependent source file is modified. In addition, it also performs heuristic analysis to determine which parts of the project need to be rebuilt, and will execute a development server after rebuilding. This greatly tightens the programmer feedback cycle, as the developer need only save a modified file in order for the build system to recompile the required sources and launch a new server with the changes in place, in real time.

4.1.3 Homogeneous deployment and development

One of the project risks identified in the planning stage was the risk of complex deployment of the compiled website. This is a common problem with web development projects, where it is necessary to take a program developed on a local machine and transfer it to run on a server which often has a wildly different environment and hardware. Mitigation of this risk was achieved by combining the use of the Autotooled build system with the use of Heroku to host the website. Heroku[2] is a company which offers a hosting service whereby a repository can be uploaded and an arbitrary script executed. This means that a build script can be executed after uploading in order to compile all of the web sources on the server side, ensuring that differences in the environment can be compensated for by the build system. This simultaneously mitigates one of the key project risks, while providing a homogeneous build system which offers full control of the configuration of the deployment server and local development servers.

4.2 Test automation

A stated goal of the project objectives is that there should be adequate automated test coverage of any APIs. This is achieved by using a combination of unit tests of individual software components and black box testing of the entire web server using controlled datasets.

4.2.1 png: Generation of test data

The generation of test data is achieved using a novel application of Markov chains for the purpose of creating plausible data from existing datasets. A Markov chain is a system used in statistical modelling of stochastic processes, in which the transitions between each state of a process are modelled as a function of their probability [29]. When applied to a body of text, a set of Markov chains can model the probability that a given word may follow a preceding word. Listing 4.2 shows a JavaScript implementation of this text

[2]Heroku | Cloud Application Platform `https://www.heroku.com/`

parsing, building up an array of Markov chains (`wordTrees`), as well as recording each start and end word for sentences.

```
1  for (var i = 0; i < list.length; i++) {
2    var sentence = list[i].split(' ');             // Split line into words
3
4    startwords.push(sentence[0]);                  // Record first word
5    terminals.push(sentence[sentence.length - 1]); // Record last word
6
7    for (var j = 0; j < sentence.length - 1; j++) {
8      var current = sentence[j], next = sentence[j + 1];
9
10     if (wordTrees[current] === undefined)
11       wordTrees[current] = [next];                // Create a word tree
12     else
13       wordTrees[current].push(next);              // Add word to tree
14   }
15 }
```

Listing 4.2: Markov chain implementation in JavaScript.

By picking a random starting word from the set of states, it is possible to generate a syntactically correct sentence by traversing the states by picking a random transition from one state to the next, based on the probability of it occurring in the training set. See Listing 4.3 for a JavaScript implementation of a Markov text generator. This method of text generation has applications where a seemingly plausible body of text must be generated from an input text, and has been used largely only for novelty purposes or for the generation of spam. The Plausible Nonsense Generator (png) is a tool which uses a Markov text generator to parse an input CSV training input, and generates a CSV of the same format with psuedo-random values for individual fields.

The Plausible Nonsense Generator proved extremely valuable for the purposes of black-box systems testing, where it was possible to generate test datasets by training with PIP-DB. This provided useful test data while mitigating the risk of exposing the confidential PIP-DB data inadvertently.

```
1  var next = function() {
2    var word = rand(startwords); // Pick a random starting word
3    var sentence = [word];       // Create a sentence array
4
5    while (wordTrees[word] !== undefined) {
6      var nextWords = wordTrees[word];
7
8      word = rand(nextWords);
9      sentence.push(word);
10
11     if (terminals[word] !== undefined)
12       break;
13   }
14
15   return sentence.join(' ');
16 };
```

Listing 4.3: Markov text generator implementation.

4.2.2 Test coverage using branch analysis

Unit tests are used to provide fine grained testing of individual software components. The purpose of these tests are to ensure correctness of code and algorithms, and operate at a lower level than whole system black box testing. Several criteria have been proposed for assessing the adequacy of unit tests, including an analysis of the coverages of code branches. For a given unit, the percentage of control transfers which are tested can be used as a metric to indicate how thorough the testing is [30]. If every possible branch in a unit's program flow is tested, then it is said to have 100% test coverage.

In practice, achieving 100% branch coverage with unit tests becomes increasingly impractical with software of significant complexity. Nested conditional logic results in an infinite set of unique paths through a program, requiring a structured approach to determine which paths to prioritise in testing and which can be ignored [31]. In pip-db, the Clojure testing framework is used to write unit tests, and the third party library Cloverage[3] is used to perform automatic branch analysis of the source code. Figure 4.3 shows the generated report for a given source file. The output is colour coded for each line of source input such that green shows total test coverage, yellow indicates partial coverage and red represents untested logic. These visual indicators of test coverage allow for a quick review of test adequacy, and ensure that testing efforts can be focused on critical code sections, thus achieving the project objective of providing comprehensive automated test coverage of the API.

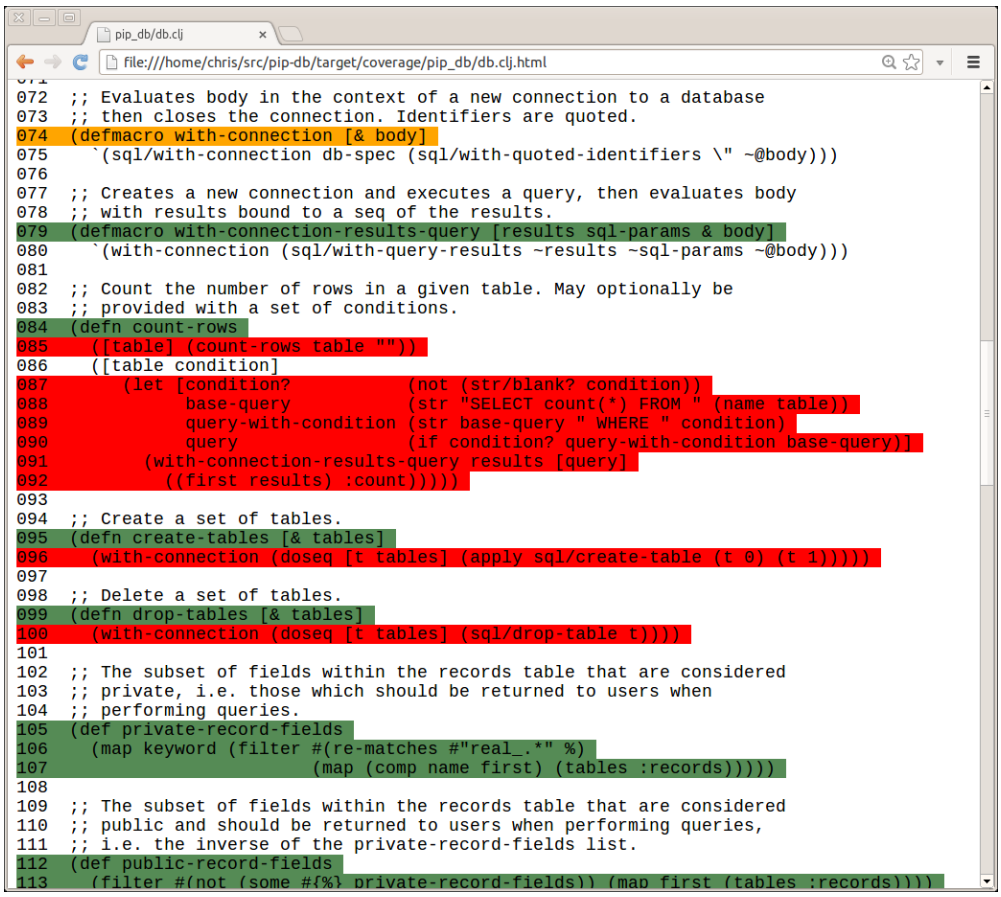


Figure 4.3: Screenshot of test coverage report.

[3]Cloverage `https://github.com/lshift/cloverage`.

4.2.3 Continuous integration

The usefulness of testing can only be asserted if tests are executed regularly during development. This is the basic tenant of the Test Driven Development process, which mandates that development occur in very small cycles of continuously writing new tests concurrently with the main development effort [32]. One such technique for encouraging test driven development is the concept of continuous integration, in which the a cycle of development is accompanied by continuous integration of new features into the main build, in order to enable early regression catching [33, 34]. In pip-db, continuous integration testing is provided by Travis CI[4], a service which uses a similar process to Heroku as described in Section 4.1.3, automatically executing the full test suite whenever a new revision is made.

The execution of tests occurs when a revision is published, and is performed asynchronously on a dedicated test server, which performs the tests and reports back whether the tests passed or failed. This way, it is possible to be continuously developing on the main project whilst ensuring that tests are never missed. This reduces the risk of human error causing a regression to enter into the main development branch unnoticed, as well as reducing the development cycle by performing testing asynchronously on dedicated hardware.

4.3 Task automation

The third category of infrastructure tooling is task automation. Task automation tools fulfil the self evident role of automating a sequence of actions which would otherwise have to be performed manually by the developer. The value of a task automation tool can be assessed based on the frequency with which the task needs to be performed, and the amount of time saved by automating it. Two examples of task automation tools which have been written for pip-db are a dataset analysis tool and a repository manager.

4.3.1 dsa: Dataset analysis

It is important to analyse the contents of PIP-DB in order to best understand how it should be stored and interacted with, but a manual analysis of the dataset would taken hundreds of man hours due to its large size. Additionally, we would need to repeat this analysis if the dataset were to be modified. In order to mitigate the risk of a change in the dataset during development, a dataset analysis tool (dsa) was written which would perform automated statistical analysis of the dataset. The dsa tool parses an input dataset and produces a machine parsable output file containing meta data about the dataset, including such properties as the percentages of unique values for each column, a list of the most frequent values, the mean, range and mode of numerical values, and a number of other properties which prove influential in the design of the data back-end. Figure 4.4 shows a graph of one of the properties which it analyses.

The dsa tool proved valuable in the early design stages of the web server, where a broad understanding of the dataset contents led to justified design decisions in the back-end, with real data to support design decisions. It also proved useful during the development of the Plausible Nonsense Generator (Section 4.2.1), where it could be used to verify that the contents of generated datasets has similar properties to PIP-DB. This reduced the time required to prepare tests significantly.

[4]Travis CI `https://travis-ci.org/`.

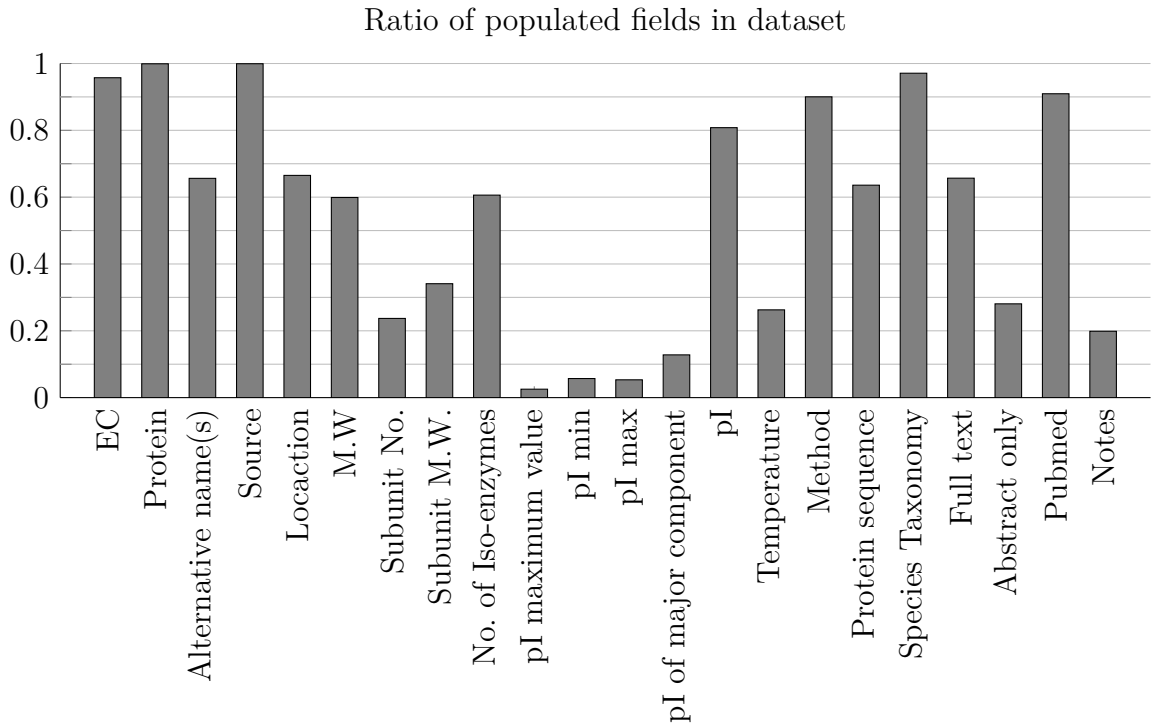


Figure 4.4: The proportion of populated keys for each field in PIP-DB.

4.3.2 pipbot: Repository management

The development of pipbot marks an intentional departure from the UNIX philosophy of creating small, single purpose programs [35], and instead attempts to create an all-encompassing tool for managing the day to day activities of repository management and maintenance. With the sole aim of automating many of the more mundane development tasks, pipbot can be invoked as either an interactive shell or a batch mode program, and has functionality for:

- **Build and deployment** Pipbot is capable of invoking the build system for local development, as well as deploying public builds to the remote web server. It can additionally generate reports of the current local and remote build configurations.
- **Generating burndowns** Pipbot can generate reports of the development activity over a given timespan, and since the last public release. This is useful for giving a quick high level overview of recent development efforts.
- **Issue tracking** Pipbot interacts with the public GitHub API in order to be able to list, show, open and close issues. This removes the need to context switch between a programming editor and a web browser during development.
- **Branching** Pipbot has an implementation of the branching model (Section 3.1.4) that allows for the creation and completion of feature branches. This enforces a consistent version control policy, making the revision history more uniform and informative.
- **Release preparation** Pipbot can create release branches and tags.

Implemented as a monolithic Python script, the pipbot tool was responsible for the huge time savings throughout the course of the project through full automation of the most common development tasks. Figure 4.5 shows an example pipbot session.

```
$ pipbot
Hello there. My name is pipbot.
-> burndown 7 days
Comparing ‘master’ against ‘master’...

  There are 56 new commits on master
  The last commit on master was 5 days, 2 hours ago
-> version
0.6.2
-> start 0.6.3
Summary of actions:
- A new branch release/0.6.3 was created, based on master.
- A new remote branch release/0.6.3 was created on origin.
- Branch release/0.6.3 tracks remote branch release/0.6.3 from origin.
- You are now on branch release/0.6.3.
- The version  number has been bumped to 0.6.3 and committed

Now, start performing release fixes. When done, use:

     pipbot finish 0.6.3

-> exit
Goodbye!
```

Figure 4.5: An example pipbot interactive session, in which the user begins a new release. User issued commands begin with the prefix "`->` ". First, the user requests a burndown of the repository activity over the past week; then they request the current version, and begin a new release with the version number `0.6.3`.

Chapter 5

Product

This chapter describes the implementation of the pip-db web application, including the web server, database back-end, and client side logic. The chapter begins with a description of the prototype implementation in order to provide context and justification for the decisions made in the final product.

5.1 Implementation of the prototype

The M1 implementation milestone culminated in the development of functional prototype, as detailed in Section 3.2.2. The purpose of the prototype was that it should offer an interactive demonstration of the final product behaviour that can be tested by potential users. As such, the primary requirement for any technology used is that it should enable the rapid prototyping of an interactive website. For this reason, the decision was made to complete the prototype implementation using the mature and stable LAMP [1] stack. Using LAMP, I was able to quickly develop a functioning website, built on top of a number of established and tested software packages. The pip-db prototype was developed by balancing the traits of *opportunistic programming*, which emphasises "speed and ease of development over robustness and maintainability of code" [25], with the needs for a solid and robust technical back-end.

The first task required for implementing the prototype was to produce a very simple schema to support storing PIP-DB in a set of MySQL tables. As this was an early prototype, support for dataset modifications and uploading of data were not added, so the MySQL import was handled solely from the command line. A set of PHP scripts were then written which would interact with the MySQL server and render web pages dynamically. Searching was performed through simple SQL select statements.

Despite the emphasis of speed over correctness during development, several efforts were made to engineer a quality implementation rather than simply a rushed one. PHP is a language that is notorious for encouraging bad programming practises through improper use and a number of caveats [36], and steps were taken to actively reduce the risk of creating spaghetti code. A clear Model View Controller architecture was used to separate application logic from back-end communications and front-end rendering, and a templating engine was used to separate HTML presentation code from program logic. Access to global super-variables was restricted through the use of a single static API, and classes were used to subdivide program components.

[1] LAMP: Linux, Apache, MySQL and PHP

5.2 Programming language selection

The amount of effort required to circumvent the faults of the PHP programming language resulted in a lot of wasted effort during the implementation of prototype, as well as leaving an unsatisfied feeling that I was having to work *against* the language, not with it. The decision was made to reimplement the project using a stronger programming language for the subsequent implementation iterations, and a selection of alternative languages were picked as candidates for evaluation. Table 5.1 shows a comparison of the three strongest candidate languages, selected after a period of broad research into popular web technologies.

	Year	Purpose	Programming style and paradigms
PHP	1995	Server-side web scripting	Object orientated, imperative, dynamically typed with implicit typing, reflective.
Python	1991	General purpose scripting	Object orientated, imperative, dynamically typed.
JavaScript	1995	Client-side web scripting	Prototype based, imperative, functional, dynamically typed.
Clojure	2007	General purpose compiled	Functional, recursive, concurrent, dynamically typed with optional explicit typing.

Table 5.1: A comparison of server-side programming languages.

Each language was evaluated with respect to a set of key desirable features:

1. **Brevity** Code written in the language should be concise and contain minimal boilerplate. PHP requires numerous caveat workarounds which pads out the size of source codes.
2. **Encourage reuse** The language should encourage reuse as a core part of its design. In PHP, the programmer must take extra effort to reduce the risk of namespace collisions and modified global state when sharing code.
3. **Architecturally sound** The language should have a clearly structured approach to web programming. PHP encourages scripting with global state and the intermingling of application and presentation logic, with any attempts to introduce composition requiring extra effort and diligence from the programmer.
4. **Expressive** The most subjective of the requirements, the programming language should allow for the relatively direct translation of ideas into functioning code, without the need to work around perceived limitations of the language.

Of the programming languages considered, Clojure was chosen as the most suitable. In addition to satisfying all of the selection criteria, the language offered the additional benefits of being incredibly fast as a result of it being a compiled language which executes on the JVM, and with simple support for concurrency due to the rejection of imperative programming in favour of immutable data structures and a transactional memory model [37, 38]. Another benefit is that as a young programming language, it has a dynamic

and fast paced development community, allowing for greater possibilities to contribute to existing projects. By far the most significant advantage of Clojure over PHP for the purposes of the project is that it is a very elegant language which departs from the "worse is better" philosophy of language design [39], and has a functional programming paradigm which is simultaneously challenging and rewarding to master.

5.2.1 The functional programming paradigm

Many popular programming languages that belong to the C family of languages describe computation as a set of statements that affects a program state. This is known as the iterative programming paradigm. By contrast, functional programming is a subtype of the declarative paradigm, in which computation is described as the evaluation of mathematical functions, which avoid state. In functional programming, the output of a function depends only on its inputs, which means that functional programs are easier to debug by design, since there is no possibility of global state or mutability. Additionally, by describing computation as the composition of functions instead of state machines, it is often simpler and more intuitive to achieve a sensible level of modularity, as "two features of functional programming languages in particular, higher-order functions and lazy evaluation, can contribute greatly to modularity" [40].

To give a quantitative comparison of imperative and functional styles, we will compare two implementations of the simple Fizz buzz game. Listing 5.1 contains a Java implementation, using an iterative and imperative approach. The body of the for loop is repeatedly evaluated, with the mutable variable x acting as a counter.

The functional Clojure implementation in Listing 5.2 uses several of the unique characteristics of functional programming: high-order functions, lazy evaluation, and anonymous lambda functions. The high-order function *map* is used in functional composition to apply a function over a collection. In this case, the anonymous condition function is applied over the lazy sequence generated by *range*. Note that the function contains no side effects, and since map returns a lazy sequence, the algorithm executes in $O(1)$ time complexity, rather than the $O(n)$ of the Java implementation. Only when the sequence is iterated over will the items in the sequence be evaluated, and cached for future evaluations.

```
1  for (int x = 1; x <= 50; x++) {
2      if (x % 15 == 0)
3          System.out.println("FizzBuzz");
4      else if (x % 3 == 0)
5          System.out.println("Fizz");
6      else if (x % 5 == 0)
7          System.out.println("Buzz");
8      else
9          System.out.println(x);
10 }
```

Listing 5.1: An imperative implementation of Fizz buzz in Java.

```
1 (map (fn [x] (cond (zero? (mod x 15)) "FizzBuzz"
2                    (zero? (mod x  5)) "Buzz"
3                    (zero? (mod x  3)) "Fizz"
4                    :else              x))
5      (range 1 51))
```

Listing 5.2: A functional implementation of Fizz buzz in Clojure.

5.2.2 Web programming with Clojure

In the M1 prototype implementation, a HTTP GET request from a client is handled in the following manner:

1. A HTTP request arrives and a new thread is spawned by Apache.
2. The request path (e.g. `GET /index.php`) is transformed into a file system lookup (e.g. `/var/www/index.php`).
3. The PHP interpreter is invoked and the matching file is evaluated line by line.
4. The output of the PHP interpreter is packed into a HTTP response body.
5. The HTTP response is annotated by Apache with relevant headers, including response code, cache control, content type, and timestamps.
6. The HTTP response is sent back to the client.
7. The handler thread terminates and the socket is closed.

In the above sequence, the programmer may only affect the behaviour by modifying the contents of the PHP file which is evaluated. In Clojure, the *Ring*[2] and *Compojure*[3] libraries provide a sequence for handling a HTTP request which is more minimalistic:

1. A HTTP request arrives and is transformed into a Clojure map.
2. This map is funnelled into a *ring handler*, which is a function that accepts a request map and is expected to produce a response map.
3. The response map is transformed into a HTTP response and sent back to the client.

The disadvantage of this more minimalistic behaviour is that it requires extra effort on behalf of the programmer to achieve the same behaviour as is provided by a LAMP stack. The advantage is that this minimalism provides much greater control over the process and the generation of responses. Figure 5.1 shows an example ring response map, showing a basic "Hello, world!" response.

The Clojure web stack provides no automatic concurrency, and routing is handled very differently from Apache. In Apache, the built in router automatically transforms requests into file system lookups, whereas in Clojure, the router is a function that determines which ring handler to execute based on the contents of the request map. There is therefore no default relationship between source files and accessible paths on the web server. Routing in pip-db is discussed in Section 5.3.1.

```
{
  :status   200,
  :headers {"Content-Type" "text/html;charset=UTF-8"},
  :body     "<html><body><h1>Hello, World!</h1></body></html>"
}
```

Figure 5.1: Example ring handler response map.

[2]https://github.com/ring-clojure/ring
[3]https://github.com/weavejester/compojure

5.3 Prototype rewrite

After the decision was made at the end of TP1 to forgo PHP in favour of Clojure for future development, the M1 prototype was reimplemented, requiring development of much lower level server logic and middleware.

5.3.1 Routing

In a LAMP stack, the Apache server provides a routing map which translates URLs into file paths, such that a request for the path "/foo/bar/index.html" would translate into a filesystem lookup for the file `index.html` in the subdirectory `foo/bar/`. In Clojure, there is no relationship between source code and request paths, and the programmer must define a route map which dispatches handler functions for each different path in the website. There is a set of macros for this purpose, allowing the programmer to declare specific handler functions for different HTTP methods (e.g. GET, PUT, and POST) and path combinations.

In pip-db, each page (e.g. index, login, search) is given a dedicated namespace, and each namespace contains ring handler functions named after the HTTP methods which they support. For example a GET request for the path "/login" will be handled by the function *login/GET*. The exception to this naming convention is the API namespace, which is detailed in Section 5.5.2. Each ring hangler function accepts a single *request* map, which contains the required information to generate a response, such as POST values and headers. Listing 5.3 shows the routes defined for pip-db. Note that ring handlers can handle dynamic paths, not just static paths. For example, the handler for the path `["/r/:id", :id id-re]` will handle all routes which start with "/r/" and matches the regular expression *id-re*, which will in turn match any 11 character string. For example, the ring handler will match requests for the paths "/r/12345678901" and "/r/abcdefghijk", but not for the path "/r/foo", since it is not an 11 digit character.

```
1  (defroutes routes
2    (GET  ["/"]                    [:as request] (index/GET       request))
3    (GET  ["/advanced"]            [:as request] (advanced/GET    request))
4    (GET  ["/r/:id.json", :id id-re] [:as request] (api/r         request))
5    (GET  ["/r/:id",      :id id-re] [:as request] (record/GET    request))
6    (GET  ["/d"]                   [:as request] (download/GET    request))
7    (GET  ["/s"]                   [:as request] (search/GET      request))
8    (POST ["/s"]                   [:as request] (search/POST     request))
9    (GET  ["/s.json"]              [:as request] (api/s           request))
10   (GET  ["/blast"]               [:as request] (blast/GET       request))
11   (GET  ["/login"]               [:as request] (login/GET       request))
12   (POST ["/login"]               [:as request] (login/POST      request))
13   (GET  ["/logout"]              [:as request] (logout/GET      request))
14   (GET  ["/upload"]              [:as request] (upload/GET      request))
15   (POST ["/upload"]              [:as request] (upload/POST     request))
16   (GET  ["/api/s"]               [:as request] (api/s           request))
17   (GET  ["/api/r/:id",  :id id-re] [:as request] (api/r         request))
18   (GET  ["/api/ac"]              [:as request] (api/ac          request))
19   (GET  ["/api/ping"]            [:as request] (api/ping        request))
20   (route/resources "/")
21   (route/not-found (ui/page-404)))
```

Listing 5.3: Application ring handler routes, taken from `middleware.clj`.

5.3.2 Ring handlers

The routing middleware described in Section 5.3.1 is responsible for determining which ring handler function to call based on the path in the request map. A ring handler accepts a request map and returns a response map. Each page has its own dedicated namespace in pip-db, and uses a Model View Controller architecture to ensure separation of data, application, and presentation tier logic. As a result, the ring handlers are often very simple functions which merely invoke the required view or controller component. Listing 5.4 shows the ring handlers for the upload page. In the case of a GET request, the GET ring handler invokes the *view* function, passing on the request map. For POST requests (i.e. file uploads), the file is stored in a temporary location, and the controller function *process-file* is invoked to handle the request and generate a response map.

```
1 (defn GET [request]
2   (view request))
3
4 (defn POST [request]
5   (util/with-tmp-file file ((request :params) "f")
6     (process-file file)))
```

Listing 5.4: Upload page ring handlers, taken from `pages/upload.clj`.

5.3.3 Generating HTML

Like many LISP dialects, Clojure maintains the "code as data" design which allows for powerful run time manipulation of programs by modifying and creating code before evaluating it. This feature is extremely useful for the generation of HTML, where the reverse interpretation of "data as code" allows HTML to be encoded as data structure within a program, and combined and manipulated on the fly. This removes the need for a templating engine such as was used in the M1 PHP prototype, as the data structures containing the page content can be manipulated directly.

```
1 [:body
2   [:h1 "Hello, world!"]
3   [:p {:id "foo"} "Foo"]]
```

Listing 5.5: Example Clojure representation of HTML elements.

```
1 <!DOCTYPE html>
2 <html lang="en">
3   <body>
4     <h1>Hello, world!</h1>
5     <p id="foo">Foo</p>
6   </body>
7 </html>
```

Listing 5.6: The HTML which is generated on evaluation of the Clojure example.

The lightweight Hiccup[4] library provides a set of data structures which can be used to represent HTML elements in Clojure, and provides the *html5* function for serialising these data structures as a string of HTML for sending to the client. Listing 5.5 shows a very simple web page consisting of a header and a paragraph element, and Listing 5.6 shows the HTML generated by the *html5* function for this web page.

[4]Hiccup `https://github.com/weavejester/hiccup`.

5.4 Persistent storage

The persistent storage component of the web server is responsible for storing PIP-DB in a manner that it can be organised, searched, and updated. PIP-DB consists of 5,773 records which have been collated from a number of sources by various people over the course of several years. Entries were recorded by hand, meaning that there are a number of style inconsistencies, and occasional inaccuracies as a result of human error during the data entry phase. Examples of hazards present in PIP-DB include:

- Fields which have multiple values may have the individual values separated using commas ",", slashes "/", or other delimiters.
- Numeric fields may contain imprecise, approximate, and non-numeric values (e.g. " 5", "> 10", "Room temperature").
- Fields for which no data is available are sometimes annotated as such (e.g. "N/A", "Unavailable", "-").

Each record in PIP-DB consists of 23 fields that record various properties about the protein, the experimental configuration, and cross-references to relevant external databases. Potential methods for storing PIP-DB were discussed at great length with Dr Flower, and it was made clear that the public web service should store a *verbatim* copy of PIP-DB, with no attempt made to correct for errors or inaccuracies. To account for this, a three tiered approach to data integrity was designed:

1. **Pre-processing** The first stage of data encoding involves the transcoding of PIP-DB from the current format into an intermediate representation. This involves *lossy* and *destructive* manipulation of the data, in which a set of signature based heuristics remove known null values (e.g. "N/A", "Unavailable").
2. **Serialising** The second stage of data encoding involves the non-destructive serialising of the intermediate data representation produced by pre-processing into a set of vectors which can be stored in SQL tables. This requires that the data be strongly typed, such that text is encoded as strings and numbers are encoded using appropriate integer or floating point types.
3. **Post-processing** Data post processing involves the transformation of rows within SQL tables into the required format for delivery to the user. This includes such changes as converting UNIX timestamps into human-readable dates, and appending units to numerical values.

Once the approach to data integrity had been specified, it was simply a case of developing the necessary tools to perform the required data transformations for each of the three tiers.

5.4.1 Yet Another Protein Schema

Yet Another Protein Schema (YAPS) is the unimaginatively named schema which was designed to store the PIP-DB records. After extensive analysis of the PIP-DB dataset, a set of fields were chosen to encode each record within, ensuring that they capture all pertinent information. A YAPS file format was then designed which would store these

records using JSON encoding. JSON encoding was chosen as it is a human-readable text-based file format, allowing for easy creation and manipulation.

The purpose of the YAPS file format is to capture PIP-DB records and annotate them with additional information to improve data integrity. Chiefly, basic accountability was incorporated into the file format by adding "Author" and "Source" tags which record the ID of the person who generated the dataset and source file of the dataset, respectively. A *csv2yaps* program was written which would parse a CSV encoded dataset and generate a YAPS file, providing the first tier data pre-processing functionality. Listing 5.7 shows an example YAPS file generated by *csv2yaps* that contains a single record entry.

```
1  {
2    "Encoding": "yaps",
3    "Version": 4,
4    "Date": "2014-04-20 02:29:48",
5    "Author": "chris@vm-ubuntu",
6    "Agent": "/home/chris/src/pip-db/tools/csv2yaps/csv2yaps.js",
7    "Source": "/home/chris/dataset-test.txt",
8    "No-Of-Records": 1,
9    "Records": [
10     {
11       "Protein-Names": [
12         "Acetoacetyl-CoA thiolase",
13         "Acetyl-CoA acetyltransferase"
14       ],
15       "EC": "2.3.1.9",
16       "Source": "Saccharomyces cerevisiae (Yeast)",
17       "Location": "Cytosol",
18       "MW-Min": "140000",
19       "MW-Max": "140000",
20       "No-Of-Iso-Enzymes": "1",
21       "pI-Min": "5.3",
22       "pI-Max": "5.3",
23       "Temperature-Min": "4",
24       "Temperature-Max": "4",
25       "Method": "Isoelectric focusing",
26       "Full-Text": "http://www.jbc.org/content/246/14/4424 ...",
27       "PubMed": "http://www.ncbi.nlm.nih.gov/pubmed/557183 ...",
28       "Species-Taxonomy": "http://www.ncbi.nlm.nih.gov/Tax ...",
29       "Protein-Sequence": "http://www.uniprot.org/uniprot/ ...",
30       "Sequence-Name": ">sp|P41338|THIL_YEAST Acetyl-CoA a ..."
31       "Sequence-Data": "MSQNVYIVSTARTPIGSFQGSLSSKTAVELGAVA ..."
32     }
33 }
```

Listing 5.7: An example YAPS encoded dataset, containing a single record.

Vectorisation of the YAPS file format into SQL tables involved designing a table schema which would preserve all of the encoded data while also enforcing a strict type system for numerical values. It was decided that the vectorised SQL representation of a record should include additional fields of appropriate types (*integer* and *real*) to store properties for which there *may* exist a numerical value, as well as storing the original text as a string. If a record contained a numerical value for this property, then the field was cast into that type and populated. Else, the numerical field was left empty.

For example, the "Tempearture-Min" property is stored as a string, but an additional property "real_temp_min" is created with a floating point type. A record which contains a "Temperature-Min" value of "15" will have the "real_temp_min" field populated with the value 15. However, a record with a "Temperature-Min" value of "> 15" will not have a "real_temp_min" value associated with it. This means that numerical searches can be performed by searching with the "real_temp_min" values. Any record which does not have a "real_temp_min" value will be omitted from the search. See Table 5.2 for the full SQL schema for storing YAPS encoded records.

Field	Type	Description
id*	*varchar(11)*	Unique record identifier
Protein-Names	*varchar*	Forward slash ("/") delimited names
EC	*varchar*	Enzyme commission number
Source	*varchar*	Protein source
Location	*varchar*	Organ and/or Subcellular location
MW-Min	*varchar*	Molecular weight minimum
MW-Max	*varchar*	Molecular weight maximum
Subunit-No	*varchar*	Subunit number
Subunit-MW	*varchar*	Subunit molecular weight
No-Of-Iso-Enzymes	*varchar*	Number of iso-enzymes
pI-Min	*varchar*	Isoelectric point minimum
pI-Max	*varchar*	Isoelectric point maximum
pI-Major-Component	*varchar*	Isoelectric point of major component
Temperature-Min	*varchar*	Experimental temperature minimum
Temperature-Max	*varchar*	Experimental temperature maximum
Method	*varchar*	Experimental method
Full-Text	*varchar*	URL of full text citation
Abstract-Only	*varchar*	URL of abstract-only citation
PubMed	*varchar*	URL of PubMed article
Species-Taxonomy	*varchar*	URL of species taxonomy reference
Protein-Sequence	*varchar*	URL of protein sequence reference
Notes	*varchar*	Notes and annotations
Sequence-Name	*varchar*	FASTA sequence description
Sequence-Data	*varchar*	FASTA sequence data
real_ec1	*integer*	Numerical first component of EC
real_ec2	*integer*	Numerical second component of EC
real_ec3	*integer*	Numerical third component of EC
real_ec4	*integer*	Numerical four component of EC
real_mw_min	*real*	Numerical molecular weight minimum
real_mw_max	*real*	Numerical molecular weight maximum
real_pi_min	*real*	Numerical isoelectric point minimum
real_pi_max	*real*	Numerical isoelectric point maximum
real_temp_min	*real*	Numerical temperature minimum
real_temp_max	*real*	Numerical temperature maximum
Created-At*	*timestamp*	Timestamp of dataset creation

Table 5.2: Yet Another Protein Schema definition. Fields marked with an asterisk (*) are derived indirectly, other fields may be null.

Unique record identification

It is necessary to be able to unique identify records so that individual records may be queried and selected. The M1 prototype implementation used a simple incrementing integer to assign each record to a unique counter value. This approach has several disadvantages, the greatest of which is that it facilitates mining the entire dataset using a simple web crawler. For example, if the individual record pages have the base url "/r/", then a script could be written to automatically download all ascending values in the sequence "/r/1", "/r/2", "/r/3" until it receives the first 404 file not found error, at which point it has downloaded the complete dataset. Since access to the entirety of PIP-DB is confidential, we must be able to take precautions to obfuscate the record URLs to prevent this type of crawling.

The solution implemented in pip-db is to use a unique hash to identify individual records. When a YAPS encoded record is processed, a secure SHA-1 hash of the record's values is computed, and encoded into base 64. The first 11 characters of this are then used as the unique identifier (Listing 5.8). Despite only using 11 of the checksum's characters, the low hash collision rate of SHA-1 and the base 64 encoding provides a huge amount of entropy, with over 7×10^{19} unique possible URLs.

```
1 def get_unique_identifier(record):
2     string = serialise_yaps_to_string(record)
3     hash = sha1(string)
4     b64 = base64(hash)
5     return substring(b64, 11)
```

Listing 5.8: Pseudocode for generating unique record identifiers, See `util.clj` for the Clojure implementation.

Family	Symbol	Definition
	id	Unique identifier
N	$q_0 \ldots q_n$	Any keywords
N	$a_0 \ldots a_n$	All keywords
N	$n_0 \ldots n_n$	Not keywords
N	eq	Exact phrase
P	p_l, p_h	Minimum and maximum isoelectric point
P	m_l, m_h	Minimum and maximum molecular weight
P	e_0, e_1, e_2, e_3	Enzyme commission number
P	l_s	Source location
P	l_l	Organ or sub-cellular location
P	f_n	FASTA sequence name
P	f_s	FASTA sequence string
E	m	Experimental method
E	t_l, t_h	Minimum and maximum experimental temperature

Table 5.3: Query component symbols and their definitions.

5.5 Search engine design

The search functionality of pip-db supports querying individual properties of the dataset, with the ability to combine properties to perform complex compound queries. Query properties fall into three main categories: Name queries (N), Protein queries (P), and Experimental method queries (E). Each category is composed of specific property queries. See Table 5.3 for a full description. The structure of a query Q can composed in a tree-like manner by considering Q as the sum of all individual property queries (Figure 5.2), and using a logical AND relationship to link neighbouring nodes. By treating a query as a tree structure, it can be easily mapped into LISP code by representing the root node of the tree as a set of nested lists, and then serialising the data structure into a compound SQL select statement.

A set of macros were written to create a domain specific language in Clojure for representing queries as tree structures, with a syntax for describing comparison operations (e.g. is equal, is greater than, is not equal), and compound operations for composing operations (e.g. logical and, or). The query tree can then be serialised into an SQL select statement by traversing the tree and assembling each node into a condition statement. Listing 5.9 shows the implementation of the pip-db query tree in Clojure, where logical reduction has been used to combine the AND conditionals where possible.

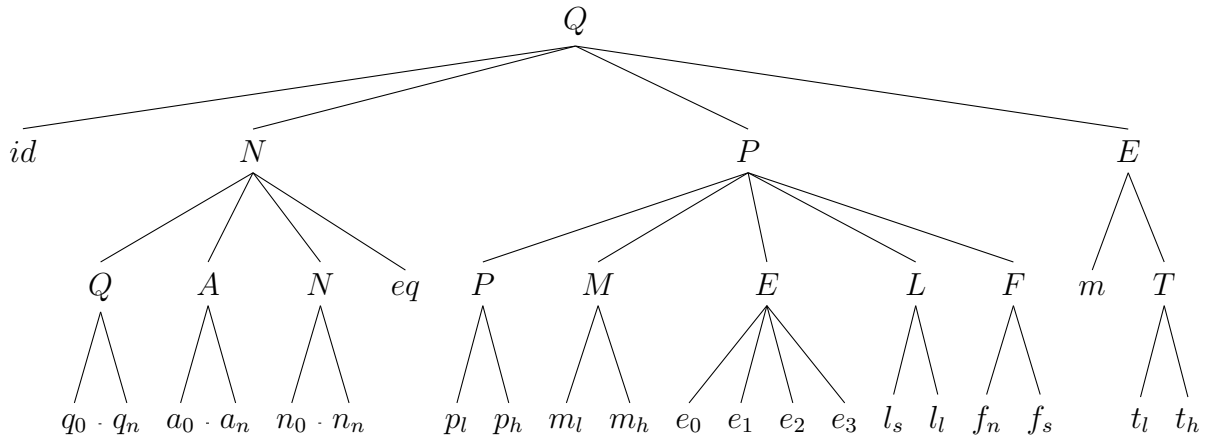


Figure 5.2: Structure of the query tree used for composing searches of PIP-DB. Leaf nodes represent properties. Nodes with an uppercase name denote compound AND conditionals.

```
1  (AND
2   (EQ   {:field "id"            :value id :exact true})
3   (for [word q]
4     (EQ {:field "Protein-Names" :value word}))
5   (for [word q_any]
6     (EQ {:field "Protein-Names" :value word}))
7   (for [word q_ne]
8     (NE {:field "Protein-Names" :value word}))
9   (EQ   {:field "Protein-Names" :value q_eq})
10  (EQ   {:field "Source"        :value q_s})
11  (EQ   {:field "Location"      :value q_l})
12  (EQ   {:field "Method"        :value m})
13  (EQ   {:field "Sequence-Name" :value seq})
14  (GTE  {:field "real_pi_min"   :value pi_l})
15  (LTE  {:field "real_pi_max"   :value pi_h})
16  (GTE  {:field "real_mw_min"   :value mw_l})
17  (LTE  {:field "real_mw_max"   :value mw_h})
18  (GTE  {:field "real_temp_min" :value t_l})
19  (LTE  {:field "real_temp_max" :value t_h})
20  (EQ   {:field "real_ec1"      :value ec1 :numeric true})
21  (EQ   {:field "real_ec2"      :value ec2 :numeric true})
22  (EQ   {:field "real_ec3"      :value ec3 :numeric true})
23  (EQ   {:field "real_ec4"      :value ec4 :numeric true}))
```

Listing 5.9: Implementation of the query tree in Clojure, from the file `query.clj`. Note the flat query hierarchy and the use of the `for` macro for expanding multivalued queries.

5.5.1 Incorporating BLAST searching

One of the extensions to the core search engine suggested by Dr Flower was to add the ability to perform queries of PIP-DB using Basic Local Alignment Search Tool (BLAST) searching. NCBI BLAST consists of a suite of search tools[5] which can query a database of proteins sequences in the FASTA format, allowing users to identify proteins using parts of sequences. Incorporating BLAST searching in pip-db involved developing a dynamic dispatcher for the search engine which would invoke the required BLAST+ executable when needed.

Dynamic dispatcher

Dynamic dispatch is the process by which a polymorphic function determines which behaviour to use at run time. The process is used extensively in pip-db, for example, routing request maps to ring handlers (Section 5.3.1). For searches, the main search function inspects the search parameters and determines whether a BLAST search is needed (i.e. whether the search parameters include a FASTA sequence). If a BLAST search is required, then the *blast/search* function is used to handle the search. Otherwise, the *db/search* function is used. In both cases, the results are merged into a results map which contains auxiliary information such as the number of records which were searched and the number of matching records. The custom HTTP headers "x-pip-db-query-terms" and "x-pip-db-records" can be used to further control the contents of the response map. See Listing 5.10 for the dispatcher implementation.

[5]BLAST+ executables `http://blast.ncbi.nlm.nih.gov`.

```
1  (defn search [request]
2    (let [sequence      ((request :params) "seq")
3          blast?        (not (str/blank? sequence))
4          params        (request :params)
5          headers       (request :headers)
6          query-terms?  (not (= (headers "x-pip-db-query-terms") "None"))
7          records?      (not (= (headers "x-pip-db-records")     "None"))
8          matching-records     (if blast? (blast/search request)
9                                           (db/search    request))
10         no-records-searched (if blast? (blast/no-of-records)
11                                          (db/no-of-records))]
12     (merge
13      (if records?
14        (let [returned-records (take max-no-of-returned-records
15                                     matching-records)]
16          {:No-Of-Records-Returned  (count returned-records)
17           :Records                  returned-records}))
18      {:No-Of-Records-Matched       (count matching-records)
19       :Max-No-of-Returned-Records   max-no-of-returned-records
20       :No-Of-Records-Searched       no-records-searched}
21      (if query-terms?
22        {:Query-Terms               (dissoc params "seq_name")})))))
```

Listing 5.10: Search handler and dynamic dispatcher. Accepting a request map, the search handler dispatches the appropriate search function (line 8), wrapping the results into a response map.

In line 8, either the *blast/search* or *db/search* functions are dispatched to handle the search request, depending on whether it is a BLAST or non-BLAST search, respectively. Listing 5.11 shows the implementation of the *db/search* function for performing non-BLAST searches.

```
1  (defn search [request]
2    (let [query-str (params->str (request :params))]
3      (map row->record (search-results query-str))))
```

Listing 5.11: The *db* namespace search function.

The *params->str* function (line 2) is responsible for serialising a parameter map into a structured query tree, and serialises it into an SQL select statement. This is used to generate a vector of SQL record rows which are mapped over the *row->record* function (line 3), converting them back into structured YAPS records by post-processing the data.

The BLAST search function (Listing 5.12) first executes the *blastp* program with an appropriate environment and input filtered from the request map (line 2), and then maps the output of the program into the function *result->records* which performs a reverse lookup of the matched sequences and performs repeated database searches for the matched records using the parameter map. Listing 5.13 shows the implementation of the *results->records* function, showing the functional composition of the db search function to produce blast search (line 3).

```
1  (defn search [request]
2    (let [results (blast-results ((request :params) "seq"))]
3      (flatten (map #(result->records request %) results))))
```

Listing 5.12: The *blast* namespace search function.

```
1 (defn result->records [request result]
2   (let [params  (assoc (request :params) "seq_name" (result :title))
3         records (db/search (assoc request :params params))]
4     (map #(wrap-blast-result % result) records)))
```

Listing 5.13: BLAST search output processing.

5.5.2 API and mobile code

After the implementation of the core search engine component, an API was designed which would expose the search functionality publicly. The purpose of a public API is to allow for programmatic public interaction with the search engine. This allows users to write their own clients for performing searches. Communications with the public API uses the JSON file format transmitted over HTTP. The reason for using JSON is that it is a lightweight data encoding which is widely and simply supported in a number of programming languages, and translates easily to and from the map structures which are used internally within the pip-db web server. All that was required to implement the public API was to assign ring handlers which wrap the internal search functions into JSON responses, shown in Listing 5.14.

The public API offers benefits to software developers who intend to develop their own pip-db clients. Additionally, it also aids in development of the core website itself. By having a solid public API which is capable of interacting with the search engine, it is possible to avoid having to perform the dynamic generation of HTML for search results within the web server, and instead transmit mobile JavaScript scripts which interact with the public API and generate the required HTML on the client side. Doing so means that the public API is the only entry point for the search engine, which has many advantages. Firstly, it creates a simpler project, since it reduces the number of the APIs. Both internal and external clients use the same consistent API. This improves maintainability, as there is only one point of change for modifications, and it reduces the code size. Using mobile code also reduces the bandwidth load on the server, as transmitted JavaScript can be cached by web clients, requiring only JSON search results to be transmitted. This increases the throughput of the server by offloading the computational expense of generating HTML from the web server to the client browser, and uses the standard distributed application practise of balancing computation across all nodes in a system, not relying on the central web server node to preform all of the computation, leading to performance bottlenecks.

The size of the client side vs. web server code bases are indicative of the amount of computation that is performed on each side. The pip-db web server consists of 1,882 lines of Clojure LISP, and the line count for client side scripts is only marginally lower at 1,713. This indicates the balance between server-side and client-side computation. The public API also enables asynchronous AJAX communications with the search engine, a feature which is exploited extensively in the usability features described in Section 5.5.3.

```
1 (defn s [request]
2   (util/json-response (search/search request)))
3
4 (defn r [request]
5   (util/json-response (search/search (util/remap-id-param request))))
```

Listing 5.14: API ring handler implementations, taken from `api.clj`.

5.5.3 API-enabled search engine usability

As described in Section 3.2.1, a human-centred approach was adopted for the design of the website. Through consistent interaction with potential users and the evaluation of common use cases, a number of attempts to innovate the user experience for bioinformatics search engines were made. For the search page, a combination of mobile code and the public API was used to implement two features to streamline the process of entering complex searches: the first is context sensitive Autocompletion of fields using a dedicated API, and the second is a widget which indicates the number of results will be returned by a given search. Figure 5.3 shows how a client browser interacts with these APIs when the user fills in a search form.

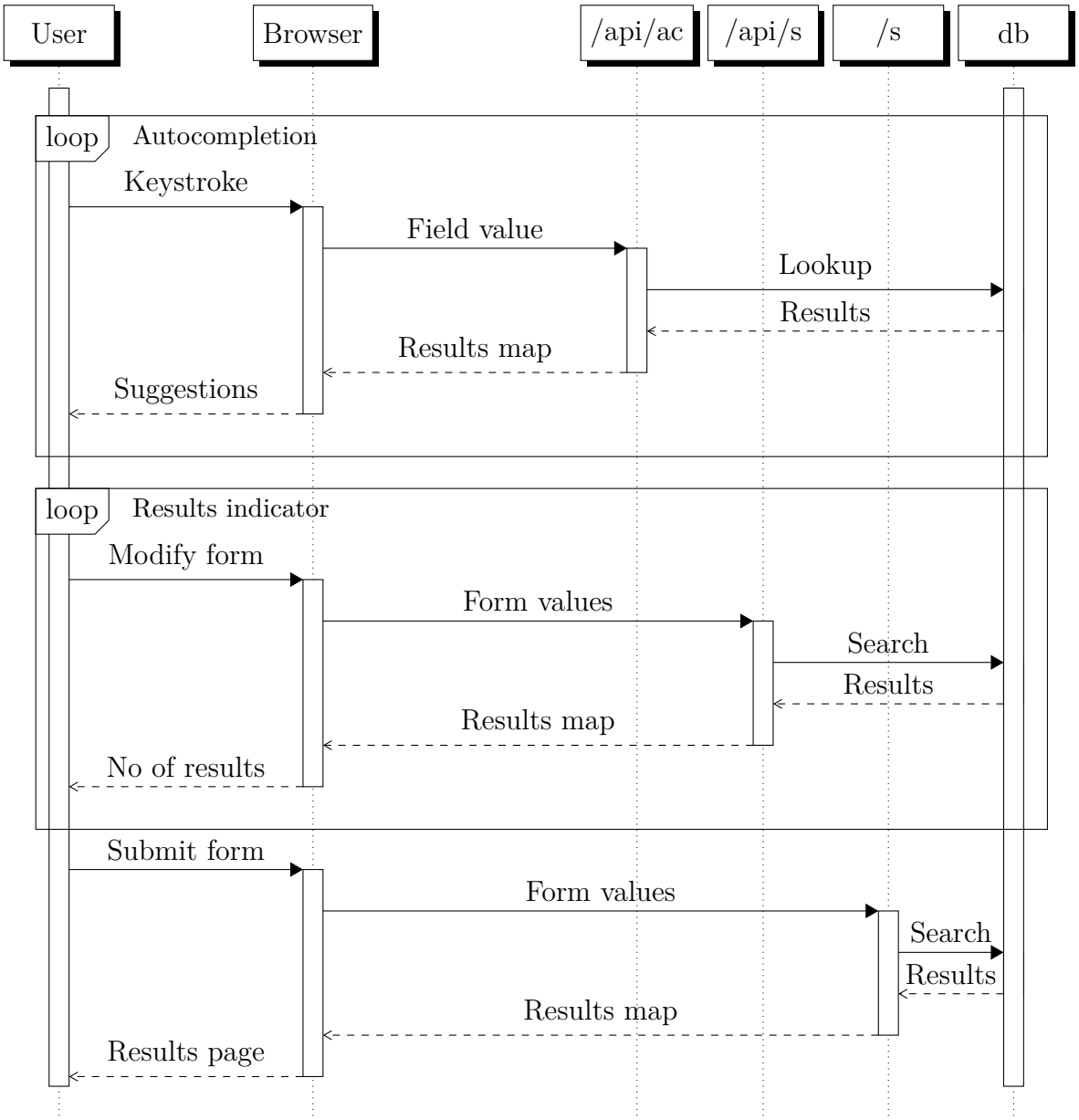


Figure 5.3: Search form sequence diagram. The instances `/api/ac`, `/api/s`, and `/s`, represent the public web services available at those locations. Communication from the browser to those services takes the form of asynchronous HTTP GET requests.

5.6 Website design and usage

This section contains an annotated visual overview of the design of the pip-db website, and its usage instructions.

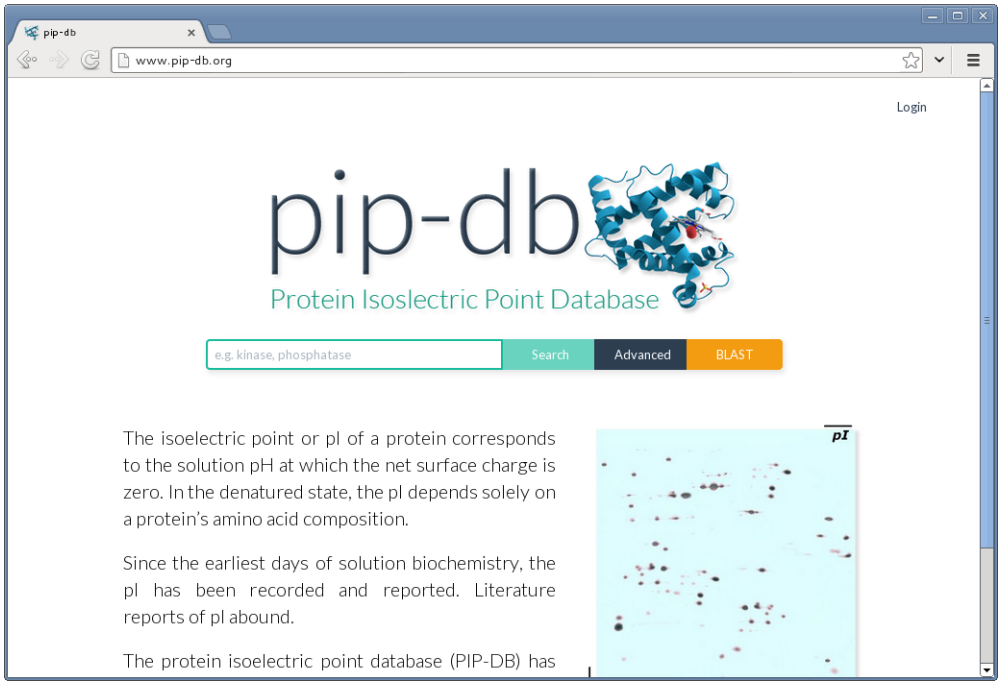


Figure 5.4: The pip-db homepage. This provides the main entry point into the website. Users may perform basic protein name queries, and navigate to the advanced search and BLAST search pages. The copy is text provided by Dr Flower to explain the purpose of PIP-DB and to provide scientific context and background for the data.

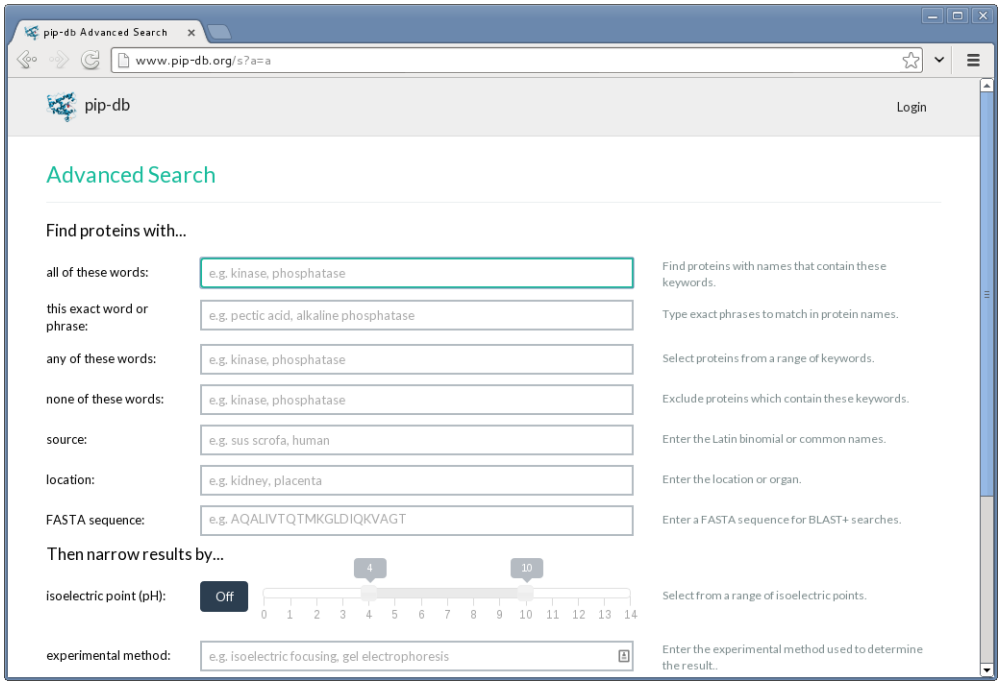


Figure 5.5: The pip-db advanced search page. This page allows users to query the database using a combination of different properties, including protein name keyword matching, and numerical properties such as temperature and pI.

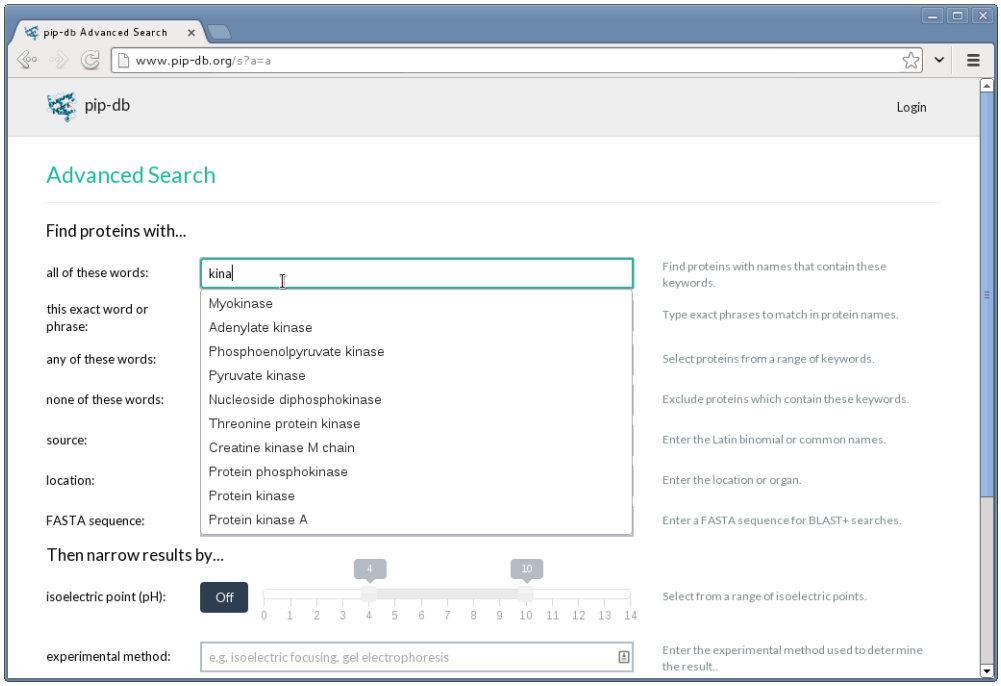


Figure 5.6: Autocompletion suggestions on the advanced search page for the term *kina*. The autocompletion API uses frequency analysis of the most common terms in the database to provide appropriate suggestions, so that the most frequently used matching terms are suggested first.

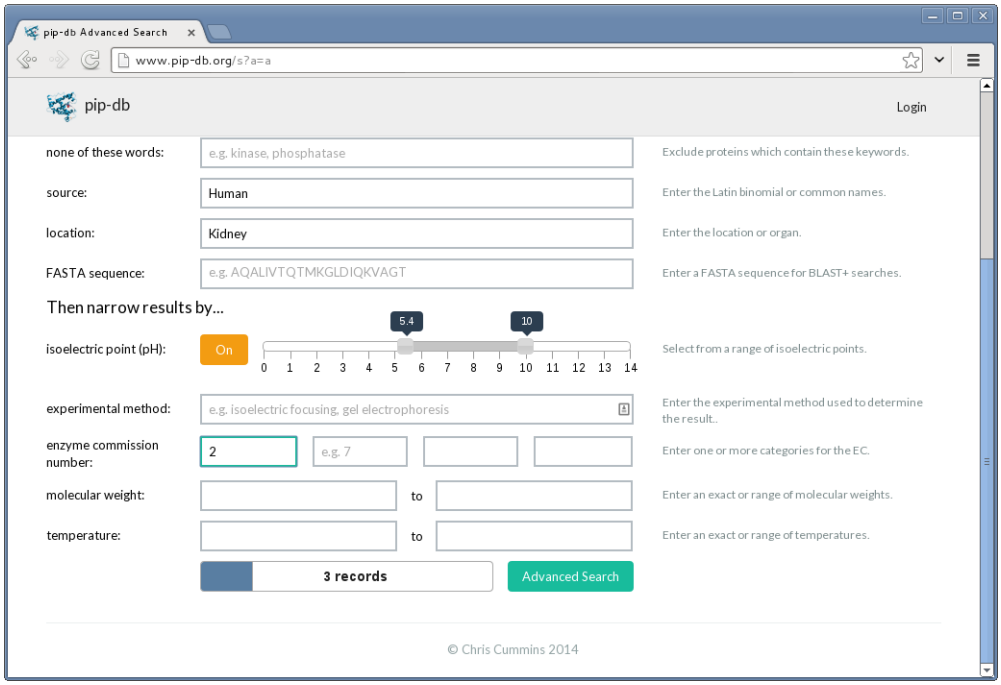


Figure 5.7: The pip-db results indicator on the advanced search page. The results indicator shows the number of records which match the current search query, and is updated dynamically as the user fills out the form. The screenshot shows that 3 records will be returned for the current search terms. Also visible is the use of a sliding range input widget for specifying a range of pI values to search.

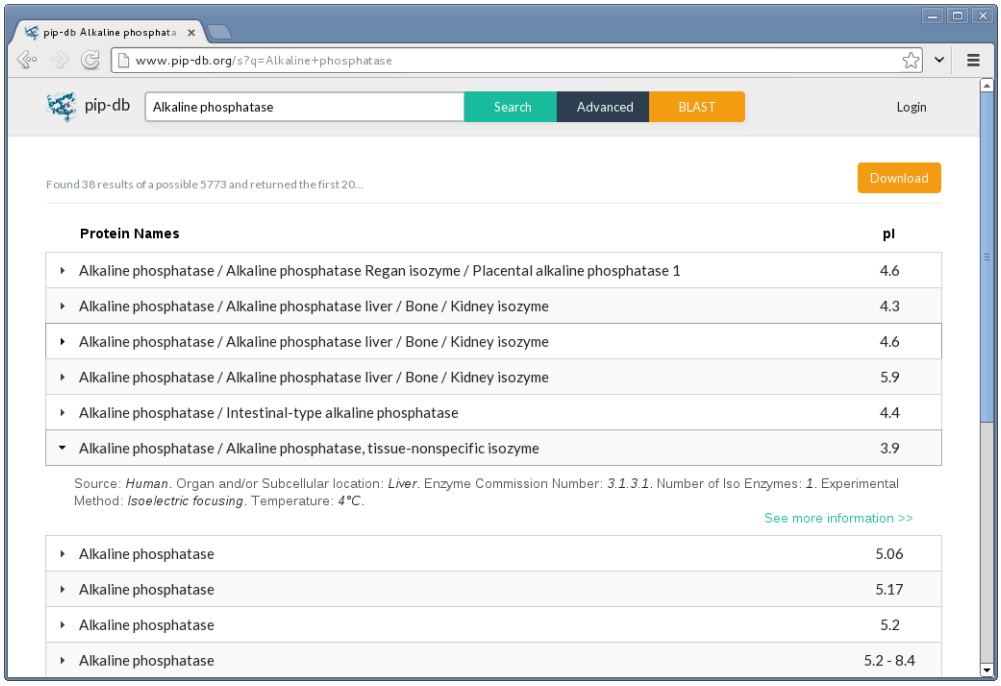


Figure 5.8: The pip-db search results page for the search term *Alkaline phosphatase*. The search results page lists the protein names and pI of each matching record. If the user clicks on a record, further information about the record is revealed. Clicking the "See more information" link will then redirect the user to that specific record's page. There is a button to download the search results in CSV or JSON formats.

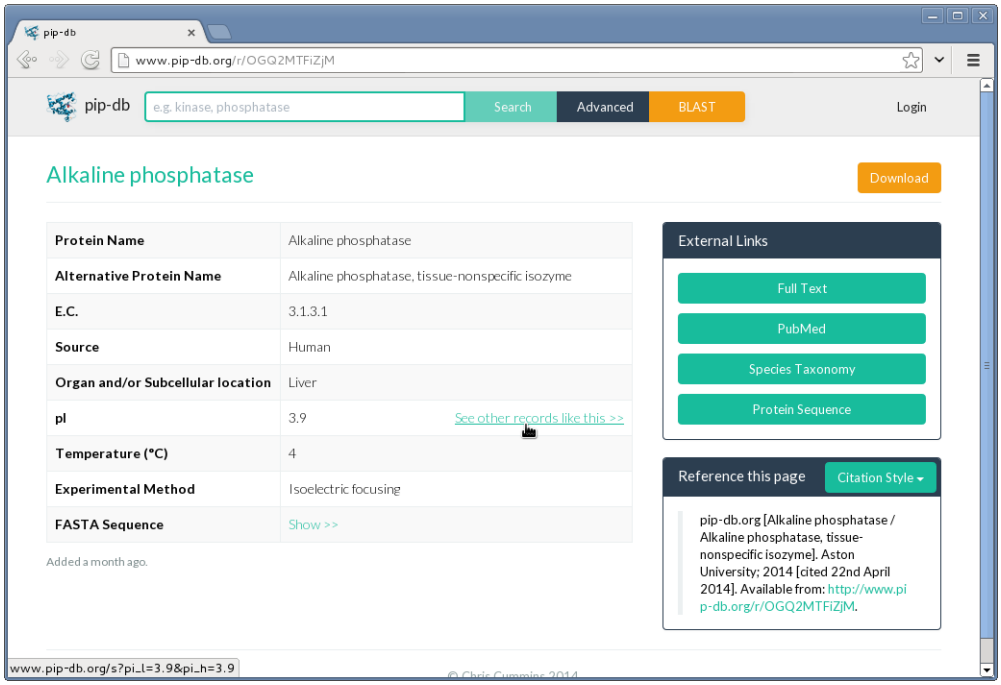


Figure 5.9: The pip-db record page, showing an *Alkaline phosphatase* entry. The page shows property values, with a "See other records like this" link for showing records with identical property values. Links and cross-references to external resources are shown in the top right, along with a button to download the record. Beneath that is an automatic page reference generator, for the purpose of citations.

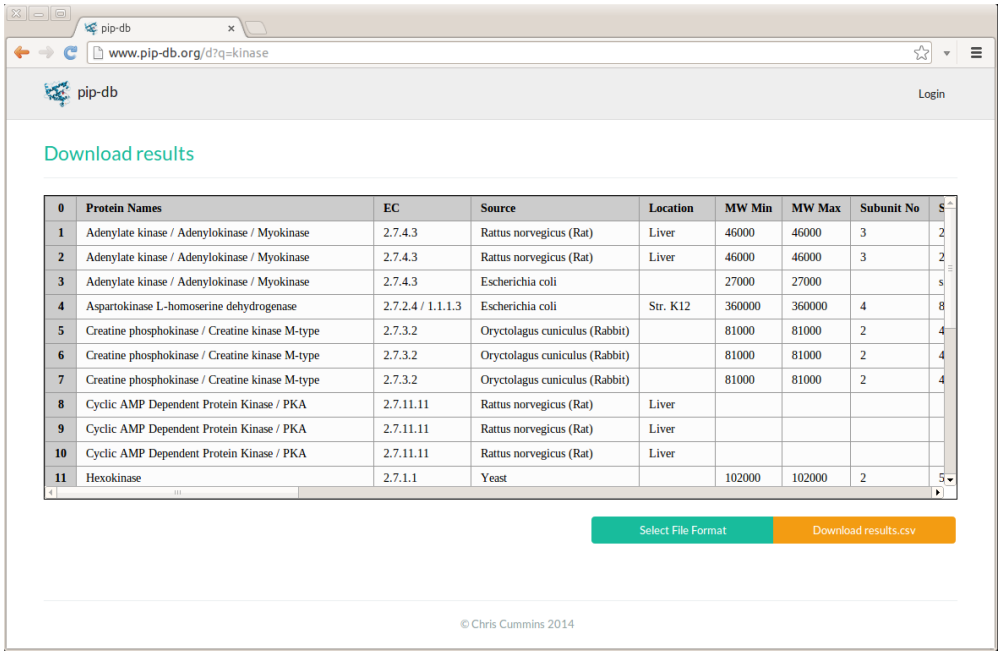


Figure 5.10: The pip-db download results page, which supports the download of search results and individual records in either CSV or JSON formats.

Chapter 6

Evaluation

Evaluation of pip-db comprised of both qualitative and quantitative evaluations. Qualitative usability testing was conducted in mid April at the end of the product development, and a criteria-based quantitative evaluation was performed at the start of May.

6.1 Usability testing

Usability testing is a popular technique in qualitative software evaluation in which participants are observed in a controlled environment performing a set of predetermined tasks in order to asses the usability of a product [41].

6.1.1 Methodology

Usability tests of pip-db were conducted over a ten day period, and consisted of presenting a set of user scenarios to a participant which they would work through. Each scenario would involve a multiple tasks designed to test the intuitive user friendliness of the pip-db website. Observations and notes are made throughout the course of the test, detailing any usability issues that the participant encountered. Appendix C describes the procedure of the tests. Appendix D contains the user scenarios which were evaluated. Five tests were conducted in total, with three PhD students specialising in a relevant biology field, and two non-scientific participants. In the case of user testing with participants who do not have a relevant scientific background, extra verbal explanation was given to provide the necessary scientific context and understanding. Following recommended practises, each test session was recorded using audio and screen capture of the testing computer [42].

6.1.2 Results

The results of user testing were generally very positive. Participants were largely satisfied with the user interface and were able to complete the tasks with little to no difficulty. However, several problems were discovered as a result of the tests:

- Multiple participants were unsure how to input a numerical range query when only an upper or lower bound is provided.
- One of the participants noticed a factual error within PIP-DB. There is no contact address to report errors and corrections.

- Every participant commented on the on the effectiveness of the results indicator on the advanced search page. However, on several occasions, the results indicator fell of sync with the search form state, leading to confusion from participants. This is because of the latency between changing a field value and updating the results indicator.
- One of the participants noticed that the only way to return to the previous page from the download results page is by using the web browser's back button.
- One of the participants was unable to locate the "Download results" button, to its placement in the top right of the screen, out of the main path of scanning.
- Multiple participants found the placeholder text very helpful in understanding the type of value which a search field expected. However, not every input field has a placeholder value, and some confusion was still had on the fields without placeholders.
- One of the participants was unsure of what query was performed when clicking the "See other records like this" button. There is no visual indicator on the search results page to show the user what the current search terms are.
- Two of the participants noted that they would prefer the ability to type in exact numerical isoelectric point query values rather than scrubbing a slider widget.

In all but one of the cases, it would be possible to mitigate the usability problem through minor modifications to the user interface. The result indicator issue is a more technically involved problem which would only be mitigated through the addition of a visual indicator to show when the value is out of sync.

6.2 Quantitative evaluation

Quantitative research involves evaluating a product using empirical and statistical techniques. For the quantitative evaluation of pip-db, a criteria-based assessment was performed using a set of criteria published by Jackson et al [43].

6.2.1 Methodology

The chosen assessment criteria offer a "quantitative assessment of software in terms of sustainability, maintainability, and usability". It was chosen as the quantitative evaluation method due to its emphasis on the evaluation of non-functional requirements, which are often overlooked through usability testing alone. The assessment criteria are worded in a manner targeted at standalone application development, not web applications. As a result, not all of the criteria were evaluated. An additional subset of the criteria was deemed irrelevant and note evaluated, for example evaluation of the current/future community.

6.2.2 Results

Appendix E contains the full table of results, with yes/no answers for each of the criteria, along with notes and annotations. The project scores strongly in the sustainability and

maintainability categories, due to it's permissive open source license, heavy use of commenting, unit testing frameworks, and automated tooling. The project scores lower in the usability category due to the relative shortage of user-orientated documentation. The importance of user documentation for websites is debatable, although at a minimum, a dedicated help page could alleviate common usability problems. The project also scores relatively lowly for testing due to its lack of automated GUI testing and scripted database testing.

Chapter 7

Conclusions

The web application described in this document entirely fulfils the stated goal of categorising and providing accessible online search functionality for PIP-DB. In addition to satisfying the core deliverables, a pragmatic approach to the development of infrastructure and tooling has resulted in the creation of several supporting projects, including the YAPS file format, pipbot repository manager, plausible nonsense generator, and CSV analyser. Additionally, contributions have been made to four popular existing open source projects as a result of development: watch-less, clojure-koans, sqlkorma, and gitstats.

The pip-db repository contains 34,205 lines of code, with an estimated development time of 605 hours, based on the 2,420 revisions in the version control history (based on an assumed average time cost of 15 minutes of development per revision). While costly in terms of time, the radical switch in programming language after the completion of the initial prototype allowed for a rare direct comparison to be made between non-trivial software written in PHP and Clojure. The comparison can be used for qualitative evaluation of the strengths of both languages, and results showed that the reimplementation in a functional language resulted in a 75% reduction in code base size for functionally identical web server implementations [44].

Development of an Autotooled build system for websites showed how shell-level parallelism could be used to reduce execution times by a factor of 5. Further work on the build system could reduce these times further by enabling parallelism within the core of automake. It is possible to repurpose the build system for future web application projects.

It my recommendation that further usability tests be performed on the existing pip-db design before any modifications are made to the user interface. While defended by some [45], it is generally believed that a usability testing sample size of 5 is too small to reveal an adequate number of usability problems [46, 47]. Further work on the pip-db website could include the development of administrative tools for modifying data, and refinements to the performance and security of the web server implementation. Increased scalability could be provided by implementing multi-threading support for request handling, and greater edge case handling could increase the server's robustness. One overlooked aspect of pip-db is the user account and registration system. The implementation of an accounts system was left incomplete as time was devoted instead to the implementation of novel features and user interaction refinements. The persistent storage component of pip-db could be the subject of future research, with NoSQL technologies finding increasing usage for the similar purposes of document storage [48–51]. Using a NoSQL store in place of the existing PostgreSQL back-end would remove the need for the vectorisation of YAPS records and greatly simply the data upload logic.

Appendix A

Risk mitigation strategies

R1 - Design is not intuitive The key to mitigation of this risk is in frequent and effective user testing and an understanding of typical and common use-cases for the product.

R2 - Project involves use of new technical skills In order to prevent this risk from having a serious impact on the project, it will be necessary to begin studying and reading about the technologies that will be used at a very early stage in the project, long before the start of the implementation.

R3 - High Level of technical complexity Avoiding this risk will involve ensuring that the scope of the project remains technically feasible, and that the software architecture is abstracted into small enough units that it is easier to focus on each one separately, as well as keeping small iterative development cycles and adequate test coverage to prevent regressions when implementing new functionality.

R4 - Complex deployment of production website A website with independent data and application logic components can result in an intricate deployment process. This is a common problem in the development of complex web application, where development and production environments must be synchronised and differences between debugging and releases builds must be accounted for. In order to mitigate this risk, a suite of tools to configure, build and deploy the website should be developed at an early stage, allowing for fast deployment of public releases.

R5 - Project milestones not clearly defined A thoroughly described and well thought out project plan will help to prevent scheduling issues and delays in development that would arise from this risk.

R6 - System requirements not adequately identified A comprehensive specification of the finished product before implementation begins will help to mitigate this risk.

R7 - Change in project requirements during development An agile approach towards accommodating for changes in the requirements should be used so as to keep the time between user feedback sessions and input from stakeholders low.

R8 - Changes in dataset format during development It is not possible to entirely avoid this risk due its nature and the dependence on third parties, but steps can be taken to prevent any delays that this would cause, chiefly, a well abstracted data parsing component which can be switched and modified if necessary to accommodate for a new dataset format.

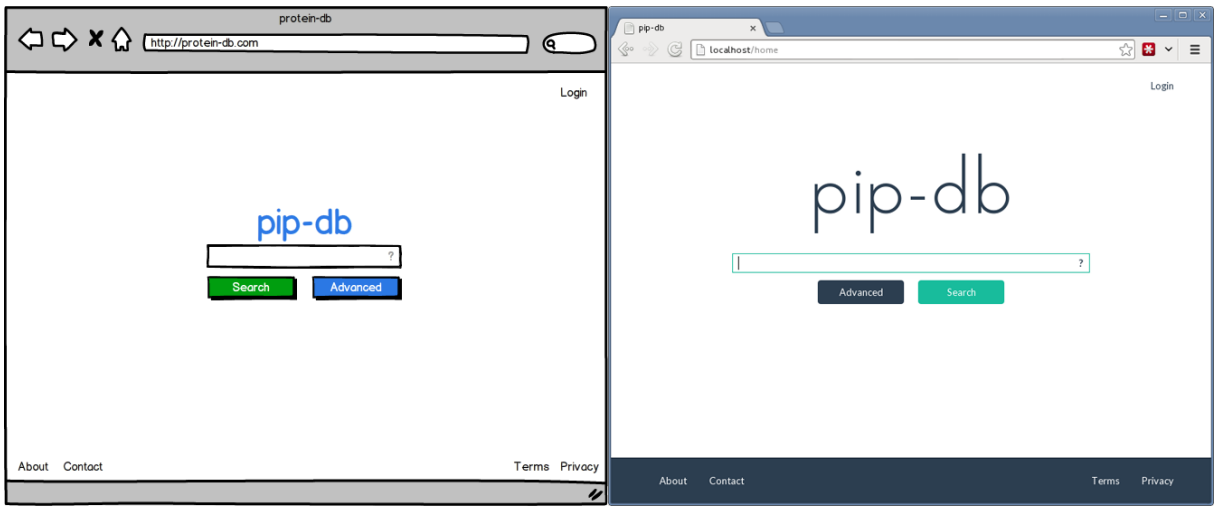
R9 - Unable to obtain required resources Since the project does not require many resources, it is important to acquire these as early on in the development process as possible, and alternative resources should be planned for, such as local test servers.

R10, R11, R12 - Users not committed to the project, lack of cooperation from users, and users with negative attitudes toward the project The usefulness of the finished project will depend largely on ensuring that the needs of the users are considered the primary goals of the design. Violating this principle may cause disillusionment from the people who are volunteering their time to assist in the project.

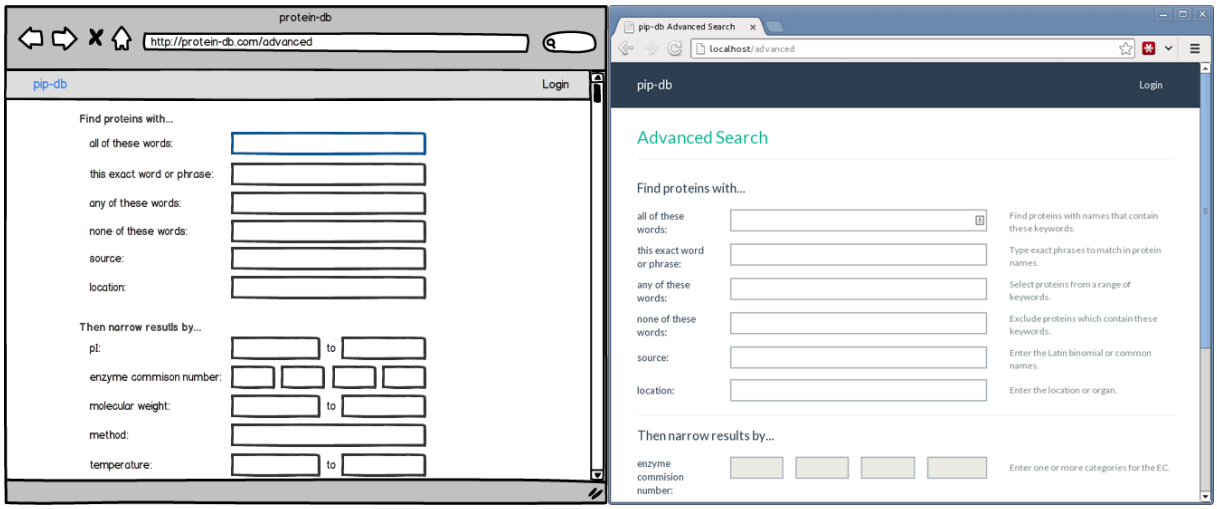
Appendix B

D1 and M1 comparison screenshots

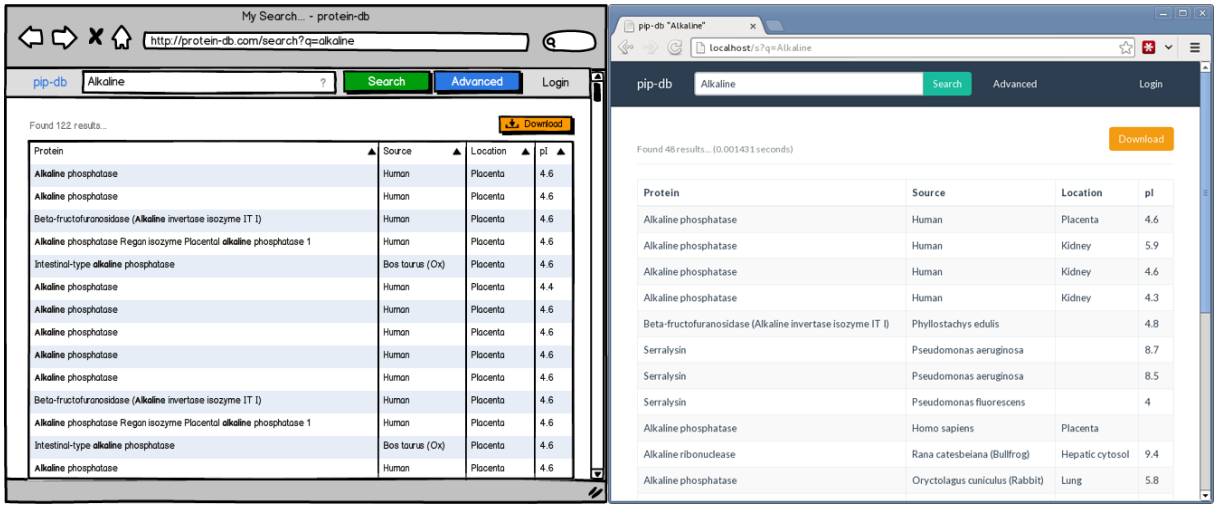
The following side by side images compare the D1 low fidelity mockups generated in Balsmiq with the M1 prototype implementation. The images on the left are static user interface mockups, the images on the right are screenshots of the interactive website.



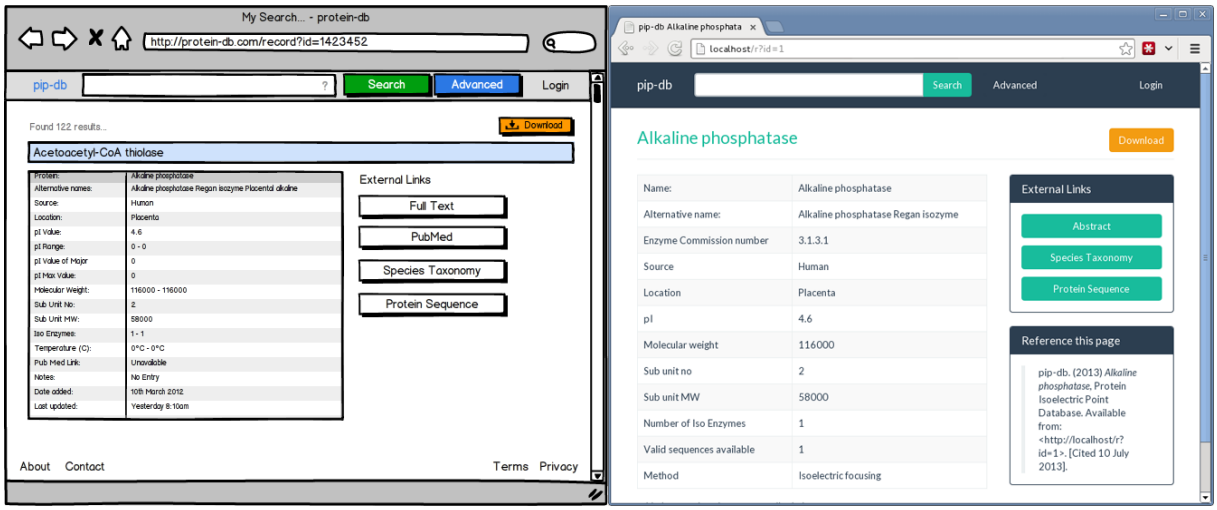
Home page.



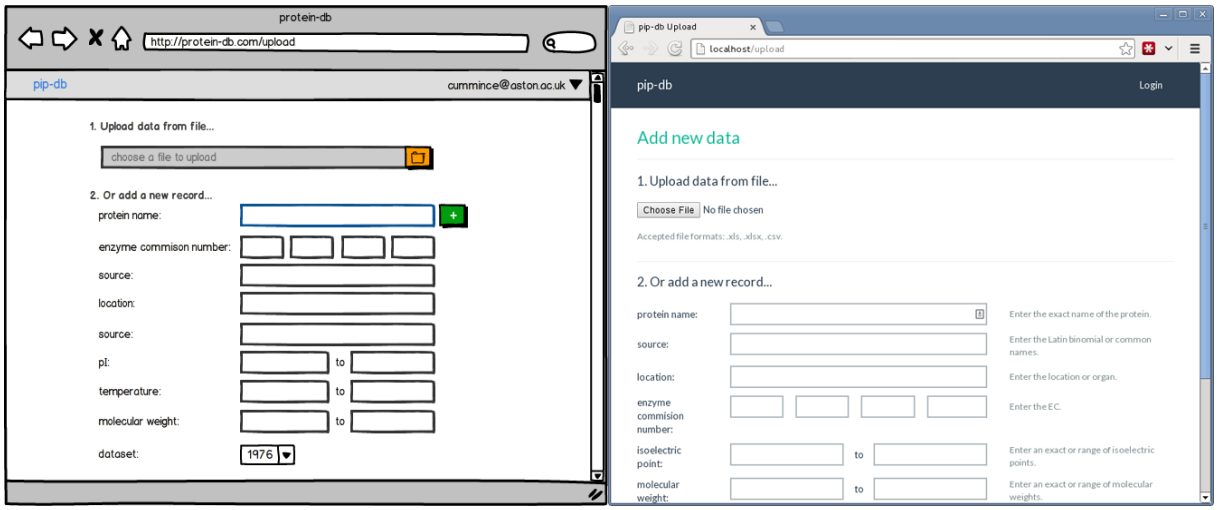
Advanced search page.



Search results page.



Record details page.



Upload page.

Appendix C

Usability testing procedure

Introduction and consent

1. **Meet participant** and introduce yourself and project.
2. Explain the **purpose of the testing** and it's role within the academic project assessment. The tests are for the product, not of the participant.
3. Request **permission to record** audio and screen capture the testing device. The purpose of the recordings are to prevent me from having to transcribe everything that is said as it is happening.
4. Request permission for recording to be **distributed as evidence** of usability testing if required.

Testing introduction

1. **Start recording**.
2. Ask for a brief overview of the **participant's background**, and what they're studying.
3. Ascertain the participant's understanding of the relevant **science background and terminology**.
4. Ascertain the participant's **familiarity with the dataset**. Any prior knowledge of the project should be stated here.
5. **Demonstrate** to the participant how to use the testing device.
6. **Thinking aloud** - provide a hands on demonstration of thinking aloud using the Aston website. Show how you would navigate to find out how to apply for a research degree in CS department.

Testing

1. **First impressions** - bring up the website on the testing device and hand over controls to participant.
2. **Present task list** to the participant and explain why it exists. Unlike many websites, pip-db is a specific tool to be used to answer questions and perform specific searches, so it is helpful to have a set of staged questions rather than letting the user browse indiscriminately.
3. **Work through tasks** - Give a spoken introduction to each one, and ask that the participant says out loud which step they are working on.

Post-test

1. What is your **overall impression** of the site?
2. Would you **use the website** for tasks like those you worked through?
3. **How likely** do you see yourself performing a task like those you worked through under normal conditions?
4. If you were to **change one thing** about the website, what would it be?
5. **Score out of 10**.

End of session

1. **Thank participant** for their time and cooperation.
2. Any closing **thoughts or questions**?
3. **Stop recording**.

Appendix D

Usability testing scenarios

The following four scenarios are to be completed by the participant during usability testing.

1. Researching a specific protein

1. You are conducting an experiment which requires you to know the isoelectric point (pI) of a protein called *Lactoferrin*.
2. You would also like to know the range of isoelectric points (lowest and highest) for all proteins obtained from the same *source* as *Lactoferrin*.

2. Performing broad searches

1. In this scenario, you would like to research *Kinase* proteins. You would like to download a CSV file which contains all proteins which match the following criteria:
    - They must contain *Kinase* in their names.
    - They were obtained from a *Human* source.
    - Their enzyme commission number begins with the three digits *2.7.1*.
    - They were discovered at a temperature greater than or equal to *4℃*.
2. Once you have downloaded the CSV file, open it in a spreasheet program and identify the one protein with a Molecular Weight of *86,000*.

3. Further broad searches

1. You would like to know the number of entries in the database which contain the word *Kinase* in their names, and compare this to the number of entries which *do not* contain the word *Kinase*.
2. From the entries which do not contain the word *Kinase*, you would like to find the protein with the *lowest* isoelectric point.
3. You would like to know the names of the authors of the PubMed article for this protein.

4. Identifying proteins using FASTA sequence

1. You have been supplied with the following protein sequence:

   ```
   >sp|P02754|LACB_BOVIN Beta-lactoglobulin OS=Bos taurus GN=LGB PE=1 SV=3
   MKCLLLALALTCGAQALIVTQTMKGLDIQKVAGTWYSLAMAASDISLLDAQSAPLRVYVE
   ELKPTPEGDLEILLQKWENGECAQKKIIAEKTKIPAVFKIDALNENKVLVLDTDYKKYLL
   FCMENSAEPEQSLACQCLVRTPEVDDEALEKFDKALKALPMHIRLSFNPTQLEEQCHI
   ```

   You would like to identify the protein that this sequence came from, and download a CSV file containing the details of all protiens which match this sequence with an isoelectric point within the range *5.1 - 5.2*.

END OF SCENARIOS.

Appendix E

Criteria-based evaluation results

This section contains the results of the quantitative criteria-based evaluation.

Criterion	**Yes/No, notes**
Descriptions of intended use cases are available	No
Documentation lists resources for further information	No
Plain-text documentation uses indentation and underlining to structure the text	Yes
API documentation documents APIs completely	Partial coverage
Documentation is held under version control alongside code	Yes
Documentation is on the project web site	Yes
Documentation on the project web site makes it clear what version of the software it applies to	Yes, documentation is version controlled
Web site has instructions for building the software	Yes
Source distributions have instructions for building the software	Yes
An automated build is used to build the software	Yes
Web site lists all third-party dependencies that are not bundled	Yes
Source distributions lists all third-party dependencies that are not bundled	Yes
Dependency management is used to automatically download dependencies	Partial
All mandatory third-party dependencies are currently available	Yes
All optional third-party dependencies are currently available	Yes
Tests are provided to verify the build has succeeded	No, although build will fail
Web site has instructions for installing the software	Yes
When an archive is unpacked, it creates a single directory with the files within	Yes
All source distributions contain a README with project name, web site, how/where to get help, version, date, license and copyright	Yes

Continued on next page...

Criterion	**Yes/No, notes**
Installers allow user to select where to install software	Yes, see configure script
Uninstallers uninstall every file or warns user of any files that were not removed and where these are	Yes
A getting started guide is provided outlining a basic example of using the software	No
Instructions are provided supporting all use cases	No
Instructions are provided supporting all use command-line, GUI and configuration options	Yes
API documentation is provided for developers	Yes
To what extent is the identity of the project clear and unique both within its application domain and generally	Partial - project description requires knowledge of domain
Software has its own domain name	Yes
Software has a logo	Yes
Software has a distinct name within its application area	Debatable
Software is trade-marked	No
Web site states copyright	Yes
To what extent is it clear who wrote the software and owns its copyright?	Stated in every copyright header
Each source code file has a copyright statement	No
Has an appropriate license been adopted	Yes, GPL v3
Software has an Open Software Initiative (OSI) recognised license	Yes, GPL v3
Project has a defined governance policy	No
Source distributions are freely available	Yes, GitHub
Source distributions are available without the need for any registration	Yes, GitHub
Anonymous read-only access to source code repository	Yes, GitHub
Ability to browse source code repository online	Yes, GitHub
Downloads page shows evidence of regular releases	Yes, there have been 57 releases. Current version is 0.6.3
Project has unit tests	Yes
Project has integration tests	Yes
Project uses automated GUI test frameworks	No
Project recommends tools to check conformance to coding standards	Yes
Project has automated tests to check conformance to coding standards	Partial
Tests are automatically run whenever the source code changes	Yes
A minimum test coverage level that must be met has been defined	No
Test results are visible publicly	Yes

Continued on next page...

Criterion	**Yes/No, notes**
Tests create their own files, database tables etc.	No
Application can be built on and run under Windows	No
Application can be built on and run under UNIX/Linux	Yes
Browser applications run under Internet Explorer	Yes
Browser applications run under Mozilla Firefox	Yes
Browser applications run under Google Chrome	Yes
Web site has page describing how to get support	Yes
Project has an e-mail address	No
Source code is structured into modules or packages	Yes
Source code structure relates clear to the architecture	Yes
Project files for IDEs are provided	No
Source releases are snapshots of the repository	Yes
There is no commented out code	Yes
There are no TODOs in the code	No, further work is suggested
Auto-generated source code is in separate directories from other source code	No
Coding standards are recommended by the project	Yes

Bibliography

[1] D. Bell. *UML basics: An introduction to the Unified Modeling Language*. IBM Corporation. 2003. URL: http://www.ibm.com/developerworks/rational/library/769.html.

[2] D. Bell. *UML basics: The sequence diagram*. IBM Corporation. 2004. URL: http://www.ibm.com/developerworks/rational/library/3101.html.

[3] F. P Brooks Jr. *The Mythical Man-Month, Anniversary Edition: Essays on Software Engineering*. Pearson Education, 1995.

[4] J. Highsmith and A. Cockburn. "Agile software development: The business of innovation". In: *Computer* 34.9 (2001), pp. 120–127.

[5] R. C. Martin. *Agile software development: principles, patterns, and practices*. Prentice Hall PTR, 2003.

[6] M. Fowler and J. Highsmith. "The agile manifesto". In: *Software Development* 9.8 (2001), pp. 28–35.

[7] R. Balduino. "Introduction to OpenUP (Open Unified Process)". In: *Eclipse site* (2007).

[8] P. Kroll and B. MacIsaac. *Agility and Discipline Made Easy: Practices from OpenUP and RUP*. Pearson Education, 2006.

[9] S. Chacon and J. Hamano. *Pro git*. Vol. 288. Springer, 2009.

[10] S. Weber. *The success of open source*. Vol. 368. Cambridge Univ Press, 2004.

[11] M. Godfrey and Q. Tu. "Evolution in open source software: A case study". In: *Software Maintenance, 2000. Proceedings. International Conference on*. IEEE. 2000, pp. 131–142.

[12] H. Chesbrough, W. Vanhaverbeke, and J. West. *Open innovation: Researching a new paradigm*. Oxford university press, 2006.

[13] E. Von Hippel. *Democratizing innovation*. MIT press, 2005.

[14] E. Raymond. "The cathedral and the bazaar". In: *Knowledge, Technology & Policy* 12.3 (1999), pp. 23–49.

[15] Free Software Foundation Inc. "Version 3". In: *GNU General Public License* (2007). URL: https://www.gnu.org/copyleft/gpl.html.

[16] K. Finley. *Github has surpassed Sourceforge and Google Code in popularity*. Blog. 2011. URL: http://readwrite.com/2011/06/02/github-has-passed-sourceforge.

[17] C. Cummins. *pip-db*. https://github.com/ChrisCummins/pip-db. Git Repository. 2014.

[18] V. Driessen. *A successful Git branching model*. Blog. 2010. URL: http://nvie.com/posts/a-successful-git-branching-model/.

[19] M. Maguire. "Methods to support human-centred design". In: *International journal of human-computer studies* 55.4 (2001), pp. 587–634.

[20] Z. Lu. "PubMed and beyond: a survey of web tools for searching biomedical literature". In: *Database: the journal of biological databases and curation* 2011.Preprint (2011).

[21] M. A. Hearst et al. "BioText Search Engine: beyond abstract search". In: *Bioinformatics* 23.16 (2007), pp. 2196–2197.

[22] K. Pavelin et al. "Bioinformatics meets user-centred design: a perspective". In: *PLoS computational biology* 8.7 (2012), e1002554.

[23] D. Bolchini et al. "Better bioinformatics through usability analysis". In: *Bioinformatics* 25.3 (2009), pp. 406–412.

[24] F. N. Egger. "Lo-Fi vs. Hi-Fi Prototyping: how real does the real thing have to be?" In: *"Teaching HCI" workshop* (2000). URL: http://www.telono.com/fr/nos-articles/lo-fi-vs-hi-fi-prototyping-how-real-does-the-real-thing-have-to-be/.

[25] J. Brandt et al. "Opportunistic Programming: How Rapid Ideation and Prototyping Occur in Practice". In: *Proceedings of the 4th International Workshop on End-user Software Engineering*. WEUSE '08. Leipzig, Germany: ACM, 2008, pp. 1–5. ISBN: 978-1-60558-034-0. DOI: 10.1145/1370847.1370848. URL: http://doi.acm.org/10.1145/1370847.1370848.

[26] T. P. Kelly. "Optimization in web caching: Cache management, capacity planning, and content naming". PhD thesis. Microsoft Research, 2002.

[27] R. Love. "Kernel korner: Intro to inotify". In: *Linux Journal* 2005.139 (2005), p. 8.

[28] I. Shields. *Monitor Linux file system events with inotify*. 2010. URL: https://www.ibm.com/developerworks/library/l-inotify/.

[29] J. Atwood. *Markov and You*. Blog. 2008. URL: http://blog.codinghorror.com/markov-and-you/.

[30] H. Zhu, P. Hall, and J. May. "Software unit test coverage and adequacy". In: *Acm computing surveys (csur)* 29.4 (1997), pp. 366–427.

[31] M. R. Woodward, D. Hedley, and M. A. Hennell. "Experience with path analysis and testing of programs". In: *Software Engineering, IEEE Transactions on* 3 (1980), pp. 278–286.

[32] K. Beck. *Test-driven development: by example*. Addison-Wesley Professional, 2003.

[33] M. Fowler and M. Foemmel. "Continuous integration". In: *Thought-Works* (2006). URL: http://www.dccia.ua.es/dccia/inf/asignaturas/MADS/2013-14/lecturas/10_Fowler_Continuous_Integration.pdf.

[34] P. M. Duvall, S. Matyas, and A. Glover. *Continuous integration: improving software quality and reducing risk*. Pearson Education, 2007.

[35] E. S. Raymond. *The art of Unix programming*. Addison-Wesley Professional, 2003.

[36] A. Munroe. *PHP: a fractal of bad design*. 2012.

[37] S. Halloway. *Programming Clojure*. Pragmatic Bookshelf, 2009.

[38] J. M. Kraus and H. A. Kestler. "Multi-core Parallelization in Clojure: A Case Study". In: *Proceedings of the 6th European Lisp Workshop*. ELW '09. Genova, Italy: ACM, 2009, pp. 8–17. ISBN: 978-1-60558-539-0. DOI: 10.1145/1562868.1562870. URL: http://doi.acm.org/10.1145/1562868.1562870.

[39] R. Gabriel. "The rise of "worse is better"". In: *Lisp: Good News, Bad News, How to Win Big* 2 (1991), p. 5.

[40] J. Hughes. "Why functional programming matters". In: *The computer journal* 32.2 (1989), pp. 98–107.

[41] J. Rubin and D. Chisnell. *Handbook of usability testing: howto plan, design, and conduct effective tests*. John Wiley & Sons, 2008.

[42] J. S. Dumas and J. Redish. *A practical guide to usability testing*. Intellect Books, 1999.

[43] M. Jackson, S. Crouch, and R. Baxter. "Software Evaluation: Criteria-based Assessment". In: *Software Sustainability Institute* (2011).

[44] C. Cummins. *Migrating PHP to Clojure*. http://chriscummins.cc/posts/migrating-php-to-clojure/. Blog. 2014.

[45] J. Nielsen. *Why you only need to test with 5 users*. 2000.

[46] J. Spool and W. Schroeder. "Testing web sites: Five users is nowhere near enough". In: *CHI'01 extended abstracts on Human factors in computing systems*. ACM. 2001, pp. 285–286.

[47] A. Woolrych and G. Cockton. "Why and when five test users aren't enough". In: *Proceedings of IHM-HCI 2001 conference*. Vol. 2. Cépadèus Toulouse„ France. 2001, pp. 105–108.

[48] B. G. Tudorica and C. Bucur. "A comparison between several NoSQL databases with comments and notes". In: *Roedunet International Conference (RoEduNet), 2011 10th*. IEEE. 2011, pp. 1–5.

[49] MongoDB Inc. *MongoDB – The Leading NoSQL Database*. Website. 2014. URL: http://www.mongodb.com/leading-nosql-database.

[50] R. Hecht and S. Jablonski. "NoSQL Evaluation: A Use Case Oriented Survey". In: *2011 International Conference on Cloud and Service Computing*. 2011, pp. 336–341.

[51] MongoDB Inc. *Top 5 Considerations When Evaluating NoSQL Databases*. Online. 2013.