Queensland University of Technology

Engineering Honours Thesis

Simplifying Programmatic Access to Bioinformatic Datasets

Author:
David Buchan-Swanson

ID Number:
n9433236

Supervisor:
Assoc. Prof. James Hogan

Unit Co-ordinator:
Prof. Margot Brereton

September 7, 2018

Generated using Pandoc and the LaTeX 2_ε typesetting system.

Note: Cyan sections remain mostly unchanged from the project proposal.

Executive Summary

Bio-informatics is an increasingly important field, and its very nature produces an extraordinarily large amount of data. Management and usage of this data becomes critical in order to achieve anything meaningful (Lee et al., 2012).

A type provider is a compiler extension for F# which allows for types to be programatically generated for large datasets. This makes them easier to manage and interact with.

The goal for this research is to simplify access to bioinformatic data, starting with Genbank. To achieve this, the aims are to explore the viability of a tool to simplify the processing of data, and to establish its usefulness in the community.

To date, an initial implementation of an F# type provider has been built, with functionality to view taxa and genomes available available on Genbank. There is continuing work on addtional functionality to reach usability by the wider community.

Included is current progress made towards completion of the type provider, challenges faced in the project and an outline of future work.

Contents

Introduction

Bio-informatics is an increasingly important field, and its very nature produces an extraordinarily large amount of data. Management and usage of this data becomes critical in order to achieve anything meaningful (Lee et al., 2012).

Being able to access this vast dataset in a simple, programmatic way will improve the workflow of interacting with Genbank. Genbank is the primary focus in the initial stages of development, as it is a large and popular dataset in the field, and there is some foundational work already completed in the space.

This paper aims to outline the methodology and research of a type provider for the F# programming language in order to more efficiently access and manipulate data, with the eventual aim of releasing a tool for the community to use.

1.1 Literature Review

1.1.1 Bioinformatics

Also known as computational biology, genetic informatics and systems biology, bioinformatics is the field of study that combines biologic data with software for analysis, storage and distribution (Lesk, 2013).

Genbank GenBank is a database managed by the National Center for Biotechnology Information (NCBI, a department of the US government) and currently contains “publicly available nucelotide sequences for 400,000 species” (Dennis A Benson et al., 2018). Submissions to the database are handled primarily online, through the BankIt system. New releases of all sequences (a public update to the database) is made every two months. From the same report delivered in 2005, Dennis A. Benson, Karsch-Mizrachi, Lipman, Ostell, and Wheeler (2005) notes that at this point in time there are only 165,000 species categorised in the database, showing continued growth even into recent times.

The similarities between different sequences in the GenBank database is one of the core ways researchers are using bioinformatics to further the general understanding of biology. (Pertsemlidis & Fondon, 2001). The Basic Local Alignment Search Tool (BLAST), another product of the NCBI, is a "sequence similarity search program", which is one of the most used of such tools available to the public (McGinnis & Madden, 2004).

Pertsemlidis and Fondon (2001) identify that sequence comparison falls into one of three main categories - identity, similarity and homology [1]. These are differentiated as follows:

- Identity refers to the repeated occurrence of *the same* nucleotide in the same position in aligned sequences
- Similarity looks only at approximate matches, and is only a useful measure of "relativeness" between them and not an absolute scoring.
- Homology is a statement of *sameness* not just of the 2 sequences in question, but of all their ancestor sequences back to a common ancestor of the two.

Given the large amount of data present, and the range of the datasets available,

[1]These are, according to Pertsemlidis and Fondon (2001), often used interchangeably, but have in fact quite different meanings.

attention is now turned to methodologies that may be employed used in management.

1.1.2 Type Providers

Type providers are a special case of the more broad "compiler extension". A type provider is specifically a compiler extension with the express purpose of generating types based on external data for the F# compiler.

There are two types of Type Provider - Generative and Erased. A generative provider produces types that are able to be compiled into a targeted assembly for use in other programs. This requires a data source that can be represented using types from the .NET family.

An erasing type provider, on the other hand, is one that can only be consumed from the project they were generated from. They are ephemeral and cannot be consumed by any other program or library. This type provider is special because it introduces the concept of *delayed members*. A delayed member allows providing types from an "infinite" [2] information space. This is useful for accessing and handling a small subset of data from a much larger one (Carter, Wenzel, & Latham, 2018).

A type provider instils a far greater trust by the programmer in the program, due to type safety.

[2]It should be noted that it is, of course, not possible to process an infinite information space. The meaning in this case is that the compiler does not limit the size of the space - a computer's available memory does.

1.1.3 Type Inference & Type Safety

Type safety is a critical component of many programming languages to ensure correctness of the programs being written. The power and sophistication of type systems are able to transform a language, and make it easier to communicate one's intentions - both to the compiler and any future readers (Cardelli, 1989). Cardelli (1989) goes on to explain that type systems adapt equally as well to all programming paradigms, including functional - the main subject of this paper in the form of F#.

Type inference takes this one step further. Duggan and Bent (1996) explain this process as a compile-time reconstruction of missing type information by analysing the usage of variables within the program. This affordance gives programmers a high degree of safety with less additional programming required.

In moving toward the context of this paper, Fages and Soliman (2006) go into depth around type inference with specific regard to systems biology. Whilst looking at a different programming language and data set (Systems Biology Markup Language and BIOCHAM, respectively), it resolves to show that analysing biochemical models using type inference provides "accurate and useful information".

1.1.4 First Class Data

Within the scope of this paper, first-class data refers to the treatment of data the same as any other construct in a program, such as functions or variables. Ideally it implies that data, especially external when discussing type providers, receives the same checks and balances that other constructs are afforded. The exact mechanics behind this, when referring to F# and external data sources, is discussed by Petricek, Guerra, and Syme (2016). In this paper, a library built by the authors is shown to perform type inference on some external data using an F# type provider. This allows the same treatment of this external data any other strongly-typed data in the program.

The library works by taking in a sample of the data, so no fixed schema is required, and analysing it to determine a general shape. This allows, for instance, the compiler to provide static analysis of some remote JSON retrieved from an HTTP request without the use of project-specific marshallers or similar.

The creator of the F# programming language presented a tutorial on the use of type providers with financial data, noting that "most financial programming and modelling is highly information rich, but our programming tools are often information sparse" (Syme, Battocchi, & Petricek, 2014). The tutorial goes on to explain how the use of type providers to turn this data into a first-class citizen of F# would "support the integration of large-scale information sources".

1.2 Research Problem

1.2.1 Research Statement

The object of the research is to make it easier to access and explore bioinformatic data. The research aims to explore the viability of a tool to enable access, and to establish the usefulness of such a tool in the wider programming community.

1.2.2 Research Questions

The main question driving all of the research is:

> How can we simplify programmatic access to large bioinformatic datasets?

Existing Project

As of the last report, there was a type provider with limited functionality. The implementation was able to download listings of taxa and genomes from the Genbank service, and provide them as types. Included was the FTP urls where the given data could be found, and a rudimentary file parser for the accession data. This acted as a proof of concept of what was to come. It enabled a basic type provider to be built and for a better scope of what was required in the future.

Updates to the Project

There have been few, if any, user facing changes to the project since the last report. Most of the changes have been to the internals of the tool itself.

3.0.1 Enhanced Logging Capabilities

In the previous version of the project, there was a logger added. However, it often did not work, was awkward to use, and was not used widely in the application.

This made it difficult to debug issues in the already notoriously difficult conditions of working with Type Providers. The following changes were complemented by far more complete instrumentation throughout the library.

Colorized Output A fairly minor addition, but an immeasurably useful one when confronted by a wall of text at a command prompt. A library, `Colorful.Console`, was used in order to print coloured text to the prompt. This included colouring `success`, `info`, `warn` and `error` logs in different colours for easy identification. The other feature was providing each logger with a new colour to print its name in. This allowed for easy identification of which module was producing which output.

A downside to adding this functionality, however, was that the distribution mechanism for the library seemed unable to pick up, or instrument, the dependency correctly when bundling for release. The net-result of this is that including the library in any other project resulted in a compiler error due to the missing dependency and the project unable to compile. This is discussed in further detail in NuGet Distribution.

Ergonomic API The previous API was not easy to use. The logger accepted only strings, which meant that there were a lot of log lines that looked as follows:

```
logger.Info(sprintf("The result of the previous call was: %s")(result))
```

With the new implementation, the `sprintf` functionality has been incorporated using `Printf.ksprintf` to enable a far more user friendly API. The same call from above now looks as follows:

```
logger.Info("The result of the previous call was: %s") result
```

This new API still allows for the same amazing functionality of F#'s type-checked `printf` statements, but reduces the verbosity of the logging statements.

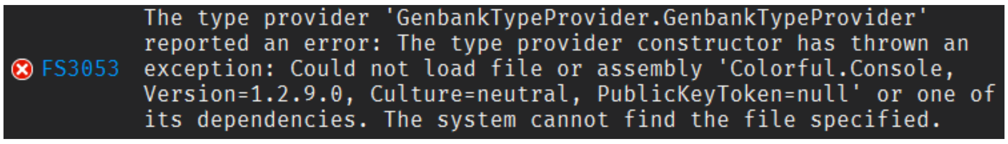
Functional file-level log writing A part of the previous logging library that never quite worked correctly was the `FileWriter` - it would cause the application to crash, or would not result in a log being created. This has been fixed to allow for the log-lines to be appended to a file, which can then be watched with `tail` as new log entries are added for a live execution summary. This makes it much easier to follow along with, as well as debug, the library when it is running in another project. Importantly, however, writing to a file causes a slowdown, as there is a non-zero cost to appending to a file, which is exacerbated by the sheer volume of logging. This could be fixed long-term by adding an in-memory buffer, and flushing it after a predetermined time. This approach would provide a "best of both worlds" result, as long as the time between buffer writes was fast enough that it did not hinder a user watching the logs.

3.0.2 NuGet Distribution

A big part of the original intention for the project was making it easy to consume by others. The primary way of consuming third-party libraries in the .NET ecosystem is using the NuGet package repository. In order to following the conventions of the community, a NuGet package was created an published. This was initially done as a proof-of-concept to ensure that it would work when required in the future.

Building the package initially proved straightforward, using a `paket.template` and the `paket pack` commands.

The first version was published and tested in a very small demo project. As soon as the newly published package was added to the example project, the compilation broke and complained of an error with `Colorful.Console`.



```
FS3053  The type provider 'GenbankTypeProvider.GenbankTypeProvider'
        reported an error: The type provider constructor has thrown an
        exception: Could not load file or assembly 'Colorful.Console,
        Version=1.2.9.0, Culture=neutral, PublicKeyToken=null' or one of
        its dependencies. The system cannot find the file specified.
```

Figure 1: The type provider errored out whenever the Colorful.Console library was used.

This meant that, not only were the provided types not being added during compilation, but there was no compilation possible. No concrete cause for the error has been found as of yet. The current work-around is to remove the `Colorful.Console` usage before packaging the application.

A fixed version was released as Version 1.0.1, and using that dependency results in the current implementation of the type provider being usable in external project. This makes consumption of the library relatively trivial, which should help aid adoption. Hurdles still to cover include macOS and Linux support to provide a truly cross-platform user experience.

3.0.3 .NET Bio

This project centres around bioinformatic data, so the obvious choice for a dependency is .NET Bio, as it includes parsers for common genomic data formats, including Genbank. This dependency was added, and a proof-of-concept was created to parse a Genbank file based on a provided type. This successfully retrieved the file, parsed it, and made the internal metadata accessible programmatically.

This was done by retrieving the ftp url of an item via the existing provided type, and then finding the correct Genbank-formatted file. Once the location of the file is determined, an FTP file download is initiated. This downloaded file is compressed using GZip, so the built-in `System.IO.Compression.GZipStream` is used by piping the FTP response stream directly into it. The `Bio.IO.GenBank.GenBankParser` from .NET Bio also accepts a stream, so the resulting output of the GZip stream is passed directly to the parser.

At present, nothing constructive is being done with this functionality. However, it was crucial to start this work and ensure it was feasible as the rest of the the project depends on the information. Further consultation is required to determine what metadata is useful, and which types to attach attributes to. This will be a large part of the future of the project. Adding the metadata to the types will make the tool useful, and allow the data to be processed.

Problems Encountered Adding the dependency took significantly longer than expected, due to problems with the .NET Bio library. The library has not been updated for over 7 months, which in and of itself is not a problem. The problem stems from the fact that the most recent version, `3.0.0` is still in `alpha`. This version of the project must be used in order to compile for modern F#. It required changes to the build pipeline that significantly reduced understanding of the process.

Online REPLs for F#, such as .NET Fiddle support adding dependencies, which would make them a great way to quickly evaluate the project's suitability and feature set. However, .NET Fiddle, from testing, does not support `alpha`-level packages (or rather, any that deviate from the standard numbering convention).

The highest-impact change, however, is that the library no longer builds on platforms other than Windows. Reduced understanding of the build pipeline has thus far made fixing this issue impossible. To combat this, a transition to the standard F# "Project Scaffold" is planned for the future. This implements a build pipeline with FAKE (F# Make), which, from testing, results in more standardised, less platform-dependent builds. Additionally, the build configuration is written in an F#-based DSL, instead of the verbose, complex and unsightly MSBuild XML Syntax.

3.0.4 Caching

One of the bigger pieces of work to be undertaken on the project is the development of a caching solution. The idea behind the caching was to reduce the time taken for requests and processing to generate the types.

The strategy was implemented at the network layer by replacing all FTP request with requests passing through the cache. As is standard, if the URL had been previously requested, the cached result was returned. If it had not been requested, a request was made, and the result was stored in the cache for any subsequent requests.

This was implemented as an in-memory cache, as maintaining a file system cache seemed significantly more difficult, and it was thought in-memory would be sufficient for the use-case. The implementation was successful, and the cache was populated as requests were made.

However, it was quickly noticed (thanks to the logging functionality) that, while the cache was being populated, it was never being hit subsequently. It was later established that the type provider maintained its *own* cache in-memory of types that had been generated. The net result of this is that once a particular type has been used, it is never required to be generated again, so never re-calls the HTTP endpoint.

This shows that the concept of using an in-memory cache was not suitable, and a file system cache should instead be used. The file system cache will have the benefit of drastically improving subsequent boot times. On the first invocation of the compiler, the cache warm-up penalty will still apply. However, any subsequent usage will result in cached value retrieval rather than FTP requests. This also allows a smarter system around cache invalidation to be implemented. With an in-memory cache, as currently understood, as soon as the compiling program (whether the IDE or the actual compiler) is closed, the cache of types is lost. The Genbank data store is only updated every three months. It is unlikely that a user of an IDE would maintain their instance running for so long, so there would have been many wasted HTTP calls with the in-memory method. By using a file system cache, a timestamp of the last change to Genbank can be maintained along side and checked during initial start-up, effectively only ever invalidating the cache when there are actual changes to the content on Genbank. This could possibly use something like the HTTP `ETAG` to only re-download files that have actually changed, rather than invalidating the cache wholesale.

The file system cache doesn't solve every use-case, however. Currently, the type provider loads *all* children at a given layer. For example, entering `Genbank.Provider`, the entire list of available taxa are downloaded. When a taxon is selected, and the genomes queried, as with `Genbank.Provider.Archara.Genomes`, the entire list of genomes is downloaded, processed and added as children. This goes on as far down as the user continues.

This means that, in an application that uses the provider as a verification/data collection tool, and not an exploration tool, many redundant API requests will be made, and a lot of unused data will be processed. The file system cache will not help reduce the amount of data that is being processed, even if it does save on a lot of the download of data. This can likely be mitigated to a certain extent by being less "generic" about how the cache works. At present, the cache has only got FTP level calls, such as `loadFile` or `loadDirectory`. If the cache had knowledge about how to process the data, it could store the data post-processing, and cut down on the amount of processing that would need to be done on every boot.

The other major problem that this caching strategy doesn't solve is that of an ephemeral compile time environment, such as a continuous integration service. This could be solved by the end user by adopting a caching solution put in place by the continuous integration provider. This use-case also seems to be beyond the scope of this project.

Challenges

4.0.1 Wasted Network Requests

As highlighted in Caching, the provider makes a lot of redundant API calls and does a lot of unused data processing if the types used are already explicitly defined. This makes sense from the point-of-view of a type provider, because the compiler has to assert that the types actually exist in order to verify correctness of the program. It is currently unknown whether a type can be "verified" rather than "generated". That is to say, given a complete type definition like

```
Genbank.Provider.Other.Genomes.``synthetic_Mycoplasma``.``Annotation Hashes``.``
```

is it possible to verify that just that type is correct using the provider, or if every layer will have to be generated again. Given a type, the URL to lookup to verify if a type is valid can be determined by simply reversing the steps used to generate the type.

4.0.2 Cross-Platform Development and Consumption

As noted previously, it is not currently possible to build the project, or any tool using the project, on a platform outside of Windows. This limits the development of the project, and hugely limits the target audience for consumption of the tool. Establishing the problem, and fixing it, is important to ensure that the widest possible reach is available. First steps in this regard involve converting the project to the "Project Scaffold" structure and changing out the build tool to FAKE.

4.0.3 Linking of Genomes

A feature of Genbank files is that they contain a list of related genes. Ideally, these genes should link, through the types, to a list of related genomes. Establishing how to re-use types in this manner, and providing a `siblings` or similar type on any given genome would greatly improve the usage of the tool for the purposes of exploration.

Future Work

In a similar theme to the previous submission, the work of most importance is the implementation of Genbank file parsing. Whilst progress has been made in this regard, there is still a ways to go. Establishing some use-cases and deciding on relevant metadata will help guide this aspect of the work.

Again, converting the project to “Project Scaffold”, and along with that conversion, adding unit tests. Unit tests will provide guarantees to consumers that the functionality works, and will continue to work. Unit tests also allow for the project to move at a faster pace without the fear of breaking existing functionality. Included in “Project Scaffold” is the `Expecto` testing library, specifically designed for F#.

Developing out a sample application is also a must, in order to demonstrate to users exactly the power and functionality available. A start has been made on this, but it currently does nothing other than print a few string to the console.

Finding a solution to the `Colorful.Console` issue when building for NuGet will enable the build to be updated more regularly.

As spoken about in the previous project report, search functionality is an important addition that should be implemented. This will broaden the audience to those who aren't looking to explore data, but to process it. These features include loading local Genbank files, searching by genome name, accession number, or some other characteristic.

A shift in priorities away from adding small features to the provider, and making the provider easier to work on and more stable must be considered. The backlog of tasks is growing at a rate too large to complete in the given time-frame, so it is important that, at the very least, there is a strong base to continue work on in the future.

Self Reflection

Once again, time management continues to be a cause for concern. Prioritisation of tasks is often wrong. I'll spend too long working on something that has little to no benefit to the project, and neglect parts of the application that could move very quickly with a little bit of work. The logging library, as useful as it is, is a good example of this. It has probably seen close to the same amount of time as the rest of the project combined.

A general lack of understanding of the platform I'm working on makes it difficult to diagnose issues. For an experience person, it may take 10 minutes, or an hour to solve an issue, whereas it will take me hours of research to likely still not come to a good result. This is evidenced by problems in the build pipeline, though I imagine that confusion over how .NET actually compiles is not a problem unique to me.

It is very evident that the timelines laid out in the project report were blatantly inaccurate. Estimation of deadlines has never been a strong suit of mine. The abundance of red on the updated Gantt chart is evidence of that.

Project Management

The Gantt Chart in Appendix A. shows the timeline for completion of the project. The completion time estimations assume for slight overages, and any unforeseen issues can make use of the modest contingency time.

As can be seen, the project is significantly behind schedule. This is a combination of poor estimations and poor time managements, with a good measure of unexpected difficulties coming up along the way.

References

Benson, D. A. [Dennis A.], Karsch-Mizrachi, I., Lipman, D. J., Ostell, J., & Wheeler, D. L. (2005). GenBank. *Nucleic Acids Research*, *33*(Database Issue), D34–D38. doi:10.1093/nar/gki063

Benson, D. A. [Dennis A], Cavanaugh, M., Clark, K., Karsch-Mizrachi, I., Ostell, J., Pruitt, K. D., & Sayers, E. W. (2018). GenBank. *Nucleic Acids Research*, *46*(Database issue), D41–D47. doi:10.1093/nar/gkx1094

Cardelli, L. (1989). *Typeful programming.*

Carter, P., Wenzel, M., & Latham, L. (2018). Type Providers. Retrieved April 16, 2018, from https://docs.microsoft.com/en-us/dotnet/fsharp/tutorials/type-providers/

Duggan, D., & Bent, F. (1996). Explaining type inference. *Science of Computer Programming*, *27*(1), 37–83. doi:10.1016/0167-6423(95)00007-0

Fages, F., & Soliman, S. (2006). Type Inference in Systems Biology. In D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, . . . C. Priami (Eds.), *Computational Methods in Systems Biology* (Vol. 4210, pp. 48–62). doi:10.1007/11885191_4

Lee, H. C., Lai, K., Lorenc, M. T., Imelfort, M., Duran, C., & Edwards, D. (2012). Bioinformatics tools and databases for analysis of next-generation sequence data. *Briefings in Functional Genomics*, *11*(1), 12–24. doi:10.1093/bfgp/elr037

Lesk, A. M. (2013). Bioinformatics. Encyclopædia Britannica, inc. Retrieved April 16, 2018, from https://www.britannica.com/science/bioinformatics

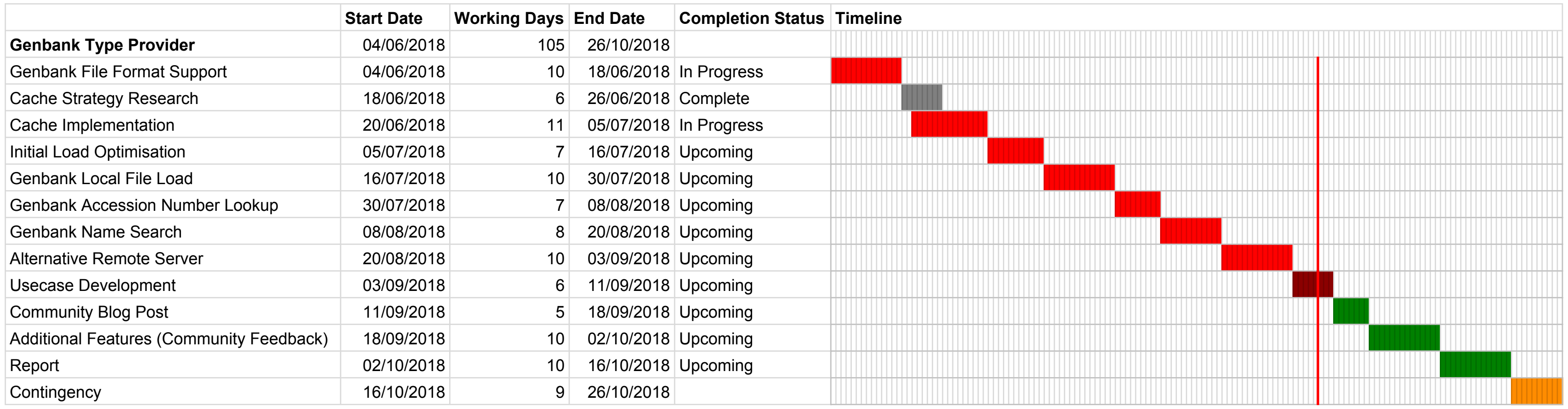
McGinnis, S., & Madden, T. L. (2004). BLAST: At the core of a powerful and diverse set of sequence analysis tools. *Nucleic Acids Research*, *32*(suppl_2), W20–W25. doi:10.1093/nar/gkh435

Pertsemlidis, A., & Fondon, J. W. (2001). Having a BLAST with bioinformatics (and avoiding BLASTphemy). *Genome Biology*, *2*, reviews2002. doi:10.1186/gb-2001-2-10-reviews2002

Petricek, T., Guerra, G., & Syme, D. (2016). Types from data: Making structured data first-class citizens in F#. (pp. 477–490). doi:10.1145/2908080.2908115

Syme, D., Battocchi, K., & Petricek, T. (2014). A World of financial data at your fingertips, functional, strongly tooled and strongly typed. In *2014 IEEE Conference on Computational Intelligence for Financial Engineering Economics (CIFEr)* (pp. xi–xi). doi:10.1109/CIFEr.2014.6924046

Appendix A | **Gantt Chart**



	Start Date	**Working Days**	**End Date**	**Completion Status**	**Timeline**
Genbank Type Provider	04/06/2018	105	26/10/2018		
Genbank File Format Support	04/06/2018	10	18/06/2018	In Progress	🟥
Cache Strategy Research	18/06/2018	6	26/06/2018	Complete	▨
Cache Implementation	20/06/2018	11	05/07/2018	In Progress	🟥
Initial Load Optimisation	05/07/2018	7	16/07/2018	Upcoming	🟥
Genbank Local File Load	16/07/2018	10	30/07/2018	Upcoming	🟥
Genbank Accession Number Lookup	30/07/2018	7	08/08/2018	Upcoming	🟥
Genbank Name Search	08/08/2018	8	20/08/2018	Upcoming	🟥
Alternative Remote Server	20/08/2018	10	03/09/2018	Upcoming	🟥
Usecase Development	03/09/2018	6	11/09/2018	Upcoming	🟫
Community Blog Post	11/09/2018	5	18/09/2018	Upcoming	🟩
Additional Features (Community Feedback)	18/09/2018	10	02/10/2018	Upcoming	🟩
Report	02/10/2018	10	16/10/2018	Upcoming	🟩
Contingency	16/10/2018	9	26/10/2018		🟧

Figure 2: Tasks in green are upcoming and are not due to have started yet, blue are in progress and within time expectations, and an orange contingency signifies between 5 and 10 days available.