The Role of Software Engineering in Bioinformatics

Brendan Sean Lawlor
Department of Computer Science, Munster Technological University, Cork, Ireland, brendan.lawlor@mycit.ie

Follow this and additional works at: https://sword.cit.ie/allthe
Part of the Bioinformatics Commons, and the Software Engineering Commons

Recommended Citation
Lawlor, Brendan Sean, "The Role of Software Engineering in Bioinformatics" (2021). Theses [online]. Available at: https://sword.cit.ie/allthe/11

This Doctoral Thesis is brought to you for free and open access by the Dissertations and Theses at SWORD - South West Open Research Deposit. It has been accepted for inclusion in Theses by an authorized administrator of SWORD - South West Open Research Deposit. For more information, please contact sword@cit.ie.
The Role of Software Engineering in Bioinformatics

Brendan Lawlor

A thesis presented for the degree of

Doctor of Philosophy

Department of Computer Science
Munster Technological University
Ireland
31 March 2021
Contents

1. Introduction 19

2. Review of Literature 26
   2.1. Software Engineering in Bioinformatics 27
   2.2. A Short History of Bioinformatics 32
      2.2.1. Definitions of Bioinformatics 33
      2.2.2. Biologists, Bioinformaticians and Computational Biologists 34
   2.3. Current Issues in Bioinformatics 36
      2.3.1. Reproducibility in Bioinformatics 36
      2.3.2. Scalability in “Big Data” Bioinformatics 42
   2.4. Research in Software Engineering 47
      2.4.1. Reproducibility in Software Engineering 48
      2.4.2. Scalability in Software Engineering 54
   2.5. Software Engineering Applied to Bioinformatics 62
   2.6. Summary 64

3. Survey of Life Science Software Development 66
   3.1. Survey Design 66
   3.2. Results 67

4. Scalability of Computation 72
   4.1. Background 73
      4.1.1. Architectural Solutions to Scalability 73
4.1.2. The Scala Language and Functional Programming . . . . . . 73
4.1.3. Akka Framework . . . . . . . . . . . . . . . . . . . . . . . . 75
4.1.4. Reactive Streams . . . . . . . . . . . . . . . . . . . . . . . . 76
4.1.5. Applications in Bioinformatics . . . . . . . . . . . . . . . . . 78
4.2. Materials and Methods . . . . . . . . . . . . . . . . . . . . . . . . 78
4.2.1. Distributed Computation in Bioinformatics using StAcS . . 78
4.2.2. Proof of Concept . . . . . . . . . . . . . . . . . . . . . . . . . 83
4.3. Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 86
4.3.1. Parallelization Over Multiple Cores . . . . . . . . . . . . . . 86
4.3.2. Parallelization Over Multiple Nodes . . . . . . . . . . . . . . 87
4.4. Discussion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 88
4.4.1. Parallelization Over Multiple Cores . . . . . . . . . . . . . . 88
4.4.2. Parallelization Over Multiple Nodes . . . . . . . . . . . . . . 88
4.4.3. Maintenance . . . . . . . . . . . . . . . . . . . . . . . . . . . . 89
4.4.4. Further Scaling . . . . . . . . . . . . . . . . . . . . . . . . . . 89
4.4.5. Further Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . 90
4.4.6. Conclusions . . . . . . . . . . . . . . . . . . . . . . . . . . . . 90
5. Scalability of Data . . . . . . . . . . . . . . . . . . . . . . . . . . . . 92
5.1. Background . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 93
5.1.1. The Structure of Data . . . . . . . . . . . . . . . . . . . . . . 93
5.1.2. Distribution . . . . . . . . . . . . . . . . . . . . . . . . . . . . 94
5.1.3. Reliability . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 95
5.1.4. Streaming . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 95
5.2. Materials . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 96
5.2.1. Apache Kafka . . . . . . . . . . . . . . . . . . . . . . . . . . . 96
5.2.2. Consumers and Producers . . . . . . . . . . . . . . . . . . . . 96
5.2.3. Partitions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 97
5.2.4. Consumer Groups . . . . . . . . . . . . . . . . . . . . . . . . . 98
5.2.5. Producers and Message Keys ........................................... 99
5.2.6. Log Compaction ......................................................... 99
5.2.7. Streaming ............................................................... 100
5.3. Methods ................................................................. 100
  5.3.1. Using Apache Kafka as a Bioinformatic Data Repository .... 100
  5.3.2. Preparation of Kafka Cluster ...................................... 101
  5.3.3. Loading data to Kafka from RefSeq ............................... 102
  5.3.4. Bioinformatic Processing .......................................... 104
5.4. Results ................................................................. 106
  5.4.1. Downloads: Benchmark vs Field of Genes ....................... 106
  5.4.2. GC Content: Benchmark vs Field of Genes ................... 107
5.5. Discussion .............................................................. 110
  5.5.1. Download .............................................................. 110
  5.5.2. GC Content ........................................................... 110
  5.5.3. Emerging Characteristics .......................................... 111
  5.5.4. Orchestration using Kubernetes .................................. 112
  5.5.5. Future Work ........................................................ 114

6. Case Study: Democratising Bioinformatics 116

  6.1. Background ............................................................ 117
  6.2. Methods ............................................................... 121
    6.2.1. Simplicity: Bioinformatic Pipelines .......................... 123
    6.2.2. Software Processes in the Service of Bioinformatics ...... 124
    6.2.3. Software Process Abstractions ................................ 125
    6.2.4. Software Process Implementations ............................ 128
  6.3. Results ............................................................... 132
    6.3.1. Collaborative Models .......................................... 132
    6.3.2. Bioinformatic Output .......................................... 134
6.4. Discussion ............................................. 145
   6.4.1. The Multidisciplinary Team .................. 145
   6.4.2. Bioinformatician as SME ..................... 145
   6.4.3. Observations .................................. 146
   6.4.4. Limitations of Reproducibility ............. 148

7. The Roles of Code ................................. 150
   7.1. The Future of Software in Bioinformatics ...... 150
   7.2. Abstraction, Subdomain, Communication ....... 154
   7.3. Mapping Code to Biology ........................ 157
   7.4. Case Study ...................................... 160
   7.5. Discussion ...................................... 161
   7.6. Further Research .............................. 163

A. Research Papers Published ..................... 187
   A.1. First Author Papers .......................... 187
   A.2. Contributing Author Papers ................. 223
   A.3. Acknowledged Contributions ................ 307

B. Scala Code Extracts ............................. 312
   B.1. Code Snippets .................................. 312

C. R Code Extracts ................................... 324
   C.1. R Code Snippets .............................. 324
Dedication

To Seán and Peggy Lawlor
With gratitude and love
From an ungrateful son.
Declaration

I hereby state that this thesis submission is entirely my own work except where otherwise accredited.

This thesis has not been submitted for an award at any other institution.

Brendan Lawlor (MTU)
17/04/2021

Prof. Roy D. Sleator (MTU)
17/04/2021
Acknowledgements

I have been lucky enough to have had the support of more than one principal supervisor during this research. Were it not for the guidance I received from my first supervisor Prof. Paul Walsh, this project would never have started. He steered me on the journey from practitioner to researcher, and I will be forever in his debt. The project would perhaps never have been finished were it not for Prof. Roy Sleator. His belief in this work, his encouragement and his insightful suggestions have deeply influenced the shape of this thesis. Paul and Roy were reliable guides through unfamiliar territory and I consider myself very fortunate to have had the chance to learn from them both.

I am grateful to the Munster Technological University and in particular to the RÍSAM PhD Scholarship Programme whose support give me the means and the confidence to undertake this research in the first place.

My thanks to the Marie Skłodowska-Curie Research and Innovation Staff Exchange Scheme for additional funding over course of this doctorate.

To Prof. Orazio Miglino and Dr. Raffaele Di Fuccio of the Natural and Artificial Cognition Lab in the University of Naples (UNINA), my thanks for their warm Italian hospitality and collegiality during my secondments to that institute.

To the management and staff, in particular Dr. Brian Kelly and Prof. Paul Walsh, of NSilico Lifescience Ltd., I’m grateful for the opportunity I was given to combine theory and practise and put my ideas to the test in that organisation.

This thesis would have been much the poorer Without the forensic attention of Dr.
Timothy Manning, and I am deeply grateful for his thoroughness and eye for detail when reviewing. My sincere thanks also to Chris Kennedy, Dr. Fraser Lauchlan, Desmond Loughrey-Grant, Ø. Macioti, Alan Rae Moat and Patrick Tinley, whose proof-reading and corrections filtered out problems than the author’s eye could have never seen. Thanks also to Greg Tynan and Dr. Olivia O’Leary for the invaluable help they gave me in preparing my defence of this thesis.

I sincerely thank the Dean and School of Graduate Studies in MTU for the availability, patience and wisdom provided, in particular during the final months of this research.

I am very grateful also to John Murphy and Eamonn Franklin of 8 West Consulting for indulging me in this, the latest in a long line of special requests.

Over these years I have had, as colleagues and coauthors, a great number of talented researchers and kind fellow-travellers, whose names can be found in the list of publications at the end of this thesis. I offer them my thanks and appreciation.

A special kind of gratitude is due to friends and family who despite years of conversation on topics that held little interest for them, still talk to me. This is especially true of my wife and children. Thank you from the bottom of my heart for your support and understanding. Normal service will shortly be restored.

As for my parents, to whom I dedicate this thesis, no words of thanks can ever be enough.
Glossary

**amino acid** The building blocks of proteins. In eukaryotes, there are 20 standard amino acids out of which almost all proteins are made. 11, 36

**Azure** Microsoft’s Cloud Computing platform. 119

**base** There are five kinds of bases from which DNA and RNA are composed: Adenine, Cytosine, Guanine, Thymine and Uracil, abbreviated A, C, G, T and U respectively. 11, 36

**cgroups** A Linux feature that controls and limits the access to resources (CPU, memory etc) of a collection of processes (for example a namespace). 52

**containerization** An operating-system-level virtualization allowing the existence of multiple isolated instances called containers. 48

**Deoxyribonucleic acid** The double-helix molecule found in all the cells of almost all living things, encoding the information required by the organism to sustain life. 36

**hypervisor** The software, firmware or hardware that creates and runs Virtual Machines (VMs). 51

**Infrastructure as Code** The representation of physical infrastructure using domain specific languages such as Terraform and Cloud Formation. 19, 113, 152
**Linux namespaces** A linux feature that allows for the isolation of processes, users, network adaptors, file systems, and so on. 51

**Next Generation Sequencing** A generic term used to describe a number of technologies that represent a second wave in genomic sequencing: faster and cheaper than the first technologies. 115

**sequence** An ordered arrangement of elements in DNA, RNA or proteins. RNA or DNA sequences are composed of bases. Proteins are sequences of amino amino acids. 43

**Whole Genome Sequencing** The sequencing of an organisms entire genome, as opposed to a targeted selection of genes using, for example, DNA Microarrays. 138
Acronyms

**AKS** Azure Kubernetes Service. 113

**ANN** Artificial Neural Network. 33

**API** Application Programming Interface. 160, 161

**AWS** Amazon Web Services. 112, 119

**BLAST** Basic Local Alignment Search Tool. 45, 82, 93, 94

**BOM** Bill of Materials. 49

**C. difficile** Clostridium difficile. 138, 139, 142

**CIT** Cork Institute of Technology. 45

**CLR** Common Language Runtime. 50

**CUDA** Complete Unified Device Architecture. 58

**DSC** Distributed Source Control. 127, 128

**DSL** Domain Specific Language. 74, 75, 164

**EaaS** Engineering as a Service. 125

**EPFL** École Polytechnique Fédérale de Lausanne. 73

**ERC** European Research Council. 73
FP  Functional Programming. 59–61, 73–75

GCP  Google Cloud Platform. 101, 104, 112, 119

GPU  Graphical Processing Unit. 155

HDFS  Hadoop Distributed File System. 44, 45

HPC  High Performance Computing. 58

IC  Integrated Circuit. 54

IDE  Integrated Development Environment. 67

INSDC  International Nucleotide Sequence Database Collaboration. 43

JNI  Java Native Interface. 85

JVM  Java Virtual Machine. 50, 74, 75, 85, 89, 148

MPI  Message Passing Interface. 58, 62, 82, 88, 90

NCBI  National Center for Biotechnology Information. 93, 94

OOP  Object-Oriented Programming. 74

OpenMP  Open Multi-Processing. 58, 82, 88, 90

OS  Operating System. 51

OSI  Open Systems Interconnection. 117

POM  Project Object Model. 49

QIIME  Quantitative Insights Into Microbial Ecology. 137

RDD  Resilient Distributed Dataset. 46, 62
**SCM** Software Configuration Management. 48–50, 64, 124, 125, 134

**SDLC** Software Development Life Cycle. 131

**SIMD** Single Instruction Multiple Data. 155

**SME** Subject Matter Expert. 5, 126, 133, 134, 138, 145–147, 153, 162

**TB** Mycobacterium tuberculosis. 138, 139, 142

**VM** Virtual Machine. 10, 51, 52, 83, 113
List of Figures

1.1. Process pyramid ............................................................... 22
1.2. Four main research themes ................................................ 24

2.1. Dollar cost of sequencing over time ................................. 44
2.2. Hadoop Distributed File System architecture ....................... 45
2.3. MapReduce word frequency example .............................. 46
2.4. Spark transformation on distributed data ......................... 47
2.5. VMs vs Containers ........................................................... 52
2.6. Dockerfiles, images and containers .................................. 53
2.7. Structure of Docker images ............................................... 53
2.8. Amdahl’s Law ................................................................. 56
2.9. Gustafson’s Law ............................................................... 57

3.1. Survey question 1 results ................................................ 68
3.2. Survey question 2 results ................................................ 69
3.3. Survey question 3 results ................................................ 70
3.4. Survey question 4 results ................................................ 71

4.1. Message Passing in Akka .................................................. 76
4.2. Thread Scheduling in Akka .............................................. 77
4.3. Smith-Waterman: Initial matrix ....................................... 79
4.4. Smith-Waterman: Scoring complete ................................ 80
4.5. Smith-Waterman: Backtracking complete .......................... 81
4.6. Worker nodes pulling work from Master Node ................................ 83
4.7. Actors on the Worker Node ..................................................... 85
4.8. Scalability over multiple nodes ............................................... 88
5.1. Producers and Consumers on a Kafka cluster ............................. 97
5.2. Anatomy of a Kafka Topic and Partitions .................................... 98
5.3. Consumer Groups ................................................................... 99
5.4. NCBI Loader agent ................................................................. 102
5.5. GC Content agent ................................................................. 105
5.6. Scalability of Downloads ......................................................... 107
5.7. Scalability of GC Content ...................................................... 108
5.8. Processing Rates of GC Content (seq/sec) ................................. 109
5.9. Suggested use context of Kafka and Akka. ................................. 115
6.1. Roles and their interfaces in bioinformatic software development ....... 120
6.2. Resistance Prediction Pipeline ............................................... 144
7.1. Sweet spot for life scientists and bioinformaticians ..................... 159
Listings

B.1. QueryBatchWorker Actor ........................................ 312
B.2. Query Subscriber Actor ........................................ 313
B.3. Database Subscriber Actor ..................................... 315
B.4. Database Sequence Subscriber Actor ......................... 317
B.5. Alignment Actor .................................................. 319
B.6. Query Collector Actor ........................................... 320
B.7. Query Batch Collector Actor .................................... 321
B.8. QuerySingleDatabaseCollector Actor ....................... 322

C.1. Survey data formatting and analysis ......................... 324
Abstract

This thesis proposes that by applying state-of-the-art software engineering tools, techniques and frameworks to currently recognised challenges in bioinformatics, improved outcomes can be attained in that field. It begins by decomposing software engineering into two categories, namely process and architecture, and choosing two key challenges in the practice of bioinformatics: reproducibility and scalability. The body of the thesis is an exploration of the intersection between these two software engineering categories and these two bioinformatics challenges. The question is asked: Can best practices in professional software engineering be applied to address key issues in the bioinformatics domain, creating positive outcomes? And can this be done without placing an extra burden on an already multidisciplinary field of study? This is answered by reasoned argument with reference to current literature, and by experiment through a series of proof of concept implementations and their related published papers. In addition, a case study is presented where software engineering processes and technologies are used in a number of bioinformatic projects, and from this a novel taxonomy of the roles of code in bioinformatics is constructed.

The conclusion is firstly that the application of software engineering elements from the two categories named above enhances reproducibility and scalability in bioinformatics, and secondly, that an intelligent integration of software engineering as a service into bioinformatics research, informed by the aforementioned taxonomy, is possible. Having demonstrated both efficacy and feasibility, this thesis concludes by recommending ways in which bioinformatic research teams move towards such integration.
1. Introduction

The relatively young field of bioinformatics is brimming with potential (Lappalainen et al. 2019). Combining the disciplines of biology and computation, it promises to bring about improvements to our quality of life, in particular through medicine, both clinical and research (Shendure et al. 2019). This promise comes from the intersection between the encoded information that underpins all life, and the technology that allows us to process that information.

Such biological codes are becoming more easily available to us, thanks to the dramatic drop in the cost of reading them. In 2003 $2.7B was spent in order to read sample human genomes (Collins et al. 2003) but today it can cost as little as $1000 to have one’s own genome sequenced, although the true cost to medical practitioners is a matter of debate (Schwarze et al. 2020).

All this is taking place against the backdrop of enormous advances in the fields of computing and software development. Access to high-speed processors, high-capacity storage and high-bandwidth networking has widened, and the specialised knowledge required to create internet-scale informatic systems has been, if not reduced, at least simplified thanks to more powerful abstractions such as containerisation, Infrastructure as Code and cloud computing (Jonas et al. 2019). In addition, our ability to develop large and complex systems has strengthened thanks to software engineering practices and tools such as agile development, distributed and integrated source control, continuous integration and other kinds of automation (Sohi et al. 2016, Vasilescu et al. 2015). This increased access is sometimes referred
to as the “democratisation of software”.

But despite all this promise and potential, there are serious problems - a crisis according to some - in bioinformatics.

Published results are proving hard to reproduce (Goodman et al. 2016, Baker 2016), and bioinformatic tools are not necessarily scaling to the available cloud infrastructure (Kashyap et al. 2015). Individual bioinformatics researchers are not always benefiting from the advances in software development mentioned above. They can spend considerable amounts of time assembling and running bioinformatics pipelines, building pipeline components from source, managing vast amounts of data on local processing platforms, and all this without necessarily being able to transmit their findings in reproducible ways.

From the perspective of a software engineer, there are some clear indications of why this might be so. While software development technology has indeed undergone a democratisation of sorts, it is not exactly the power to the people that the word might suggest. Democratisation, in this context, means that smaller teams of software developers can now compete against the larger corporations and institutions in the creation of systems that deal with vast amounts of data at high speed. While some aspects of building large-scale systems have been simplified, the development of high scale, reproducible software still requires a solid foundation of software engineering expertise, experience and culture, as evidenced by the content of curricula in cloud computing (Foster et al. 2018).

The difficulties evident in bioinformatics today can be understood in terms of a knowledge gap. The expertise required to build cloud-native systems is a specialisation within the wider software development community. Taken together with the core skills that make up software engineering it constitutes a significant body of knowledge in its own right, and is subject to rapid change and evolution.

At the same time, bioinformatics, being cross-disciplinary, has tacitly taken on the responsibility of software engineering as part of its core mission. A problem
for present-day bioinformaticians, then, is that they are expected to become their own software engineers, and without much or any exposure to the technologies, techniques or skills that would make it possible (Nguyen-Hoan et al. 2010, Wilson et al. 2014). The problem is not just the difficulty or indeed desirability of integrating yet another discipline into an already crowded multi-disciplinary field, but the fact that by and large, bioinformatic researchers have no interest in becoming software engineers - and understandably so (Wilson 2014).

Any future *democratisation of bioinformatics*, much like the software counterpart, would allow smaller teams of bioinformaticians to work with larger amounts of data and create faster pipelines. They would do this *not* by learning how to *build* cloud-native software, but by using specialised bioinformatic cloud-native tools built by software engineers, and by developing biological software in collaboration with software engineers. This thesis presents architectures, methodologies and collaborative approaches in which software engineers, armed with a working knowledge of the bioinformatic domain, will contribute towards such a future, in collaboration with bioinformaticians.

In order to bring focus to what is a broad question, the research presented here addresses the crisis in bioinformatics on two specific fronts. Firstly, on scale, it presents novel approaches for processing bioinformatic pipelines that require high degrees of processing parallelization, and which use large amounts of data. Secondly, on reproducibility, it demonstrates how software engineering techniques and technologies can allow bioinformaticians to create and transmit reproducible and verifiable research results. In both cases, these proposed solutions come from work done in software engineering research and development. In order to bring focus, two specific elements of software engineering research and practise are investigated: process and architecture.

While definitions of scalability and reproducibility are dealt with in the literature review (Chapter 2), the meanings of the terms *architecture* and *process* in the context
Components of software engineering process, and their relationships to each other.

The word *process* covers not only the activities but also the tools and practices which combine to define the environment and culture in which software development takes place (Estublier et al. 2005). It is useful to see these elements in a way that shows how they relate to, and depend on, each other, as in Figure 1.1.

Amongst the purposes of having a defined process in place is to ensure that any software developed by a team or organisation is correct, maintainable and stable (Paulk 2002).

- Correct: The software behaves as required, including considerations like performance and usability.

- Maintainable: Bugs can be fixed and new features can be added in a timely
fashion while avoiding risks to existing functionality or performance.

- Stable: The functionality provided by the software is available when needed, even under high demand and despite inevitable platform failures (hardware, network, 3rd party services).

The design, rollout and management of a software development process is a core competency of software engineering.

The term *architecture* is used in software engineering to mean the high-level or strategic design of a complex system, and typically entails decisions around the identification of sub-components of a system, and their interactions, as well as choices with respect to technologies, frameworks and platforms employed (Medvidovic & Taylor 2010). The word *design* by itself is typically taken to mean the lower-level or tactical design that is used within a given component. Software architects are expected to keep their fingers on the pulse of change and innovation in the software development field. The opensource community in particular is an ecosystem in constant evolution, with hundreds of potentially interesting projects moving through Gartner’s Hype Cycle (Linden & Fenn 2003). Experienced software architects are able to identify promising innovations from the current wave of new entries that have the utility and critical mass to survive, flourish and positively impact software systems, and then introduce these innovations appropriately into software projects and teams.

An alternative metaphor for this kind of activity is offered by Sam Newman in his book Building Microservices (Newman 2015). Newman describes the software architect as a kind of town planner, coordinating the competing needs of multiple elements in a complex system, defining zones in a way that allows for future growth.

Both architecture and design are core competencies of software engineering, along with the constant evaluation of new technologies, frameworks and platforms.

Figure 1.2 brings these four themes of scalability, reproducibility, process and architecture together into four quadrants. Each quadrant combines one of the two
selected elements of software engineering with one of the two selected issues in bioinformatics: reproducibility and scalability.

The chapters that follow will present novel research on the application of these engineering elements to these bioinformatic problems.

Each chapter is based on one or more academic publications made as part of the doctoral program that this thesis presents. The chapters are organised as follows:

Chapter 2 is a review of the relevant software and bioinformatic literature.

Chapter 3 presents findings from a survey conducted at the outset of this research (Lawlor & Walsh 2015).
Chapter 4 presents novel architectural solutions to scaling computation in bioinformatic applications (Lawlor & Walsh 2016).

Chapter 5 builds on the previous work, and presents novel architectural solutions to scaling in both computation and data by using Apache Kafka as a data repository (Lawlor et al. 2018).

Chapter 6 describes the application of software architecture and process in the development of multiple bioinformatic pipelines, and outlines the resulting publications (Lawlor & Sleator 2020).

Chapter 7 introduces a novel taxonomy to describe the roles of code in biology and, using this taxonomy, brings the above threads together in a series of conclusions and an outline of future work.
2. Review of Literature

This chapter is an updated version of the first bioinformatics paper published as part of this research under the title Engineering Bioinformatics: Building Reliability, Performance and Productivity into Bioinformatics Software (Lawlor & Walsh 2015). That paper reviewed the literature on software engineering practices in the bioinformatic research community, and surveyed bioinformatics practitioners, comparing their processes and practices with software engineering professionals.

- Citations: 13
- Views: 2271
- Altmetric: 16

This review of the relevant literature outlines the role of software engineering in bioinformatics today. It traces the history of bioinformatics, exploring tensions between biology and software. Problems currently experienced by bioinformatics researchers, in particular in relation to issues of reproducibility and scalability, are then reviewed. Relevant research in the field of software engineering is outlined,
again in reference to how that research impacts on reproducibility and scalability. Finally, suggested approaches to the application of software engineering best practices to the field of bioinformatics are examined.

2.1. Software Engineering in Bioinformatics

There is a significant lack of software engineering practices in bioinformatics when compared to commercial software development, which prevents the bioinformatic community from benefiting from decades of engineering efficiencies, rigour and quality. Software engineering skills are lacking, as is evident in the way in which software is developed in bioinformatic contexts. Although biologists and especially bioinformaticians possess programming skills, and use those skills as part of their day to day work, they do so in a way that is unstructured and not in line with modern standards of software engineering (Verma et al. 2013, Baxter et al. 2006). The problem has serious consequences for the field of bioinformatics.

Inability to Reproduce Findings: A lack of software engineering infrastructure and techniques means that many publications which use programs to process data cannot make that software or data available in a reproducible way for peer review. As a consequence, a significant number of findings are likely to be reversed or withdrawn from publication (Merali 2010). The use of infrastructure such as source code control systems and command-line build tools would improve the situation, by giving researchers the ability to easily publish and share the software that was used as part of their work. But these tools are either unknown or simply considered unnecessary for small teams by bioinformatic researchers (Lawlor & Walsh 2015).

Unreliability of Findings: All surveys on scientific software development we have reviewed cite a lack of software testing as being a constant theme of scientific development. Segal (2007) points out that the “lack of any disciplined testing procedure” is a characteristic of any development practice where the end user is also
the developer.

According to a review by Morris (2008) “unit tests often do not exist”. Because of the fundamentally important role of such tests in separating problems in the code from problems in the hypotheses, findings based on insufficiently tested software must be considered in turn insufficiently tested themselves. Equate this to the use of defective or uncalibrated lab equipment in order to fully appreciate the gravity of the problem.

**Limitations in Data Sample Size**: Many scientists run their software on multi-core desktops but do so in a single-threaded way which creates performance bottlenecks (Prabhu et al. 2011). This is most likely due to a lack of familiarity with the kind of parallel computing techniques available to software engineers. The constraints that this practice inevitably imposes on sample size or sophistication of data analysis are clear: In order to execute programs to completion on desktops, even in a time frame of hours and days, researchers will naturally reduce the number of sample points used, or eliminate steps which might increase statistical power but which have exponential or factorial performance profiles (Prabhu et al. 2011). Where multi-threaded implementations *are* used in scientific programming, they typically involve using OpenMP (for multi-core) or MPI (for multi-server) (Basili et al. 2008). These solutions use low-level primitives and as such are painstaking to develop and can result in error-prone code which is difficult to change, especially in large systems (Schindewolf et al. 2012). Software engineering research has more recently concentrated on using higher abstractions which result in more intuitive ways to achieve concurrency, for example through the use of the Actor pattern (Agha et al. 1997). There are examples of the successful porting of such engineering to the bioinformatics community (Wiewiórka et al. 2014).

**Slowing the Discovery Cycle**: Bioinformatic research is an iterative process in which the computational element takes up a significant percentage. If a researcher has to wait days to see computational results which will decide the next direction
that the research is to take, momentum is lost and the entire process of research itself
is slowed down. Software engineers can bring skills like performance optimisation
and concurrent programming to bear on this problem, significantly reducing waiting
times.

According to Prabhu et al. (2011) “a considerable portion of [scientists’] time is
spent in many tedious [software development] activities” such as converting data
formats or retro-fitting inherited software to work for new conditions. This is a
direct consequence of insufficient software engineering infrastructure and practices
around the research team. Researchers are obliged to repeatedly cobble together
solutions for every new direction they take. Naturally the nature of these improvised
solutions does not facilitate their reuse - they typically don’t exhibit high levels of
maintainability or build-reproducibility - and so the problem perpetuates itself.

In all of the above cases, we can discern a parallel to the argument made by
Ioannidis (2005) with respect to inexpert use of statistics in studies. The danger to
progress in bioinformatics is that much research may later be found to be invalid
due to inexpert or non-transparent development of software. As Verma et al. (2013)
point out, “the end goal of creating accurate and reliable scientific software is no
less critical [than with commercial software] since incorrect results would greatly
compromise the validity of the discovery”.

As *in silico* experiments become an increasingly important form of research and
development, problems of reproducibility and reliability will become more obvious
and more urgent. Moreover, software engineering techniques will be key not just in
addressing those problems, but in the initial conception and design of such experi-
ments.

These are some of the things that can go wrong in bioinformatic research when
we fail to address the problem of its software engineering deficit. But why does this
deficit arise in the first place? And what can be done to improve matters?

A number of the previously cited authors offer explanations and remedies for the
problems described above. Hannay et al. (2009) identify a general lack of formal education and training and a reliance instead on informal learning from peers. Segal & Morris (2008) among others emphasize the differences between scientific and commercial software development. Verma et al. (2013) suggest that bioinformatics represents a “unique situation for the field of software engineering”, citing issues such as a lack of a formal requirements-gathering process in bioinformatics. Umarji et al. (2009) focus exclusively on the gaps in the education of bioinformatic software developers in software engineering principles.

From the previous paragraph we can see that there are some elements in common in the way previous authors have understood the problem, and so in the solutions that they have proposed.

Some authors have found that bioinformaticians lack the necessary training in software engineering skills. Umarji et al. (2009) have surveyed bioinformatics curricula in the United States and found that “out of a total of 79 program offerings, there were only 2 instances where a software engineering related course was a required part of the curriculum” and that “there was no mention of the role and importance of software engineering in the curricula”.

The wrong processes - or no processes at all - are being applied to the practice of bioinformatic research. Verma et al. (2013) report that “little emphasis is paid on the organization and requirement gathering process in the early stages of the software”.

According to some authors, the field of scientific software development is so far removed from the commercial settings in which modern software engineering has emerged, that the rules from the latter simply do not apply. Authors have suggested that the two contexts are “fundamentally different” for reasons of subject domain complexity, requirements volatility and budgetary constraints. These differences make it problematic to “impose software engineering techniques on scientists” (Segal & Morris 2008).
Segal & Morris (2008) assert that in the case of scientific software development the subject matter is simply too complex for the “average developer”. In a similar vein, Hannay et al. (2009) suggest that “developers are much less likely to need to be domain experts” in ”regular” software development compared to scientific.

According to Segal & Morris (2008), “full up-front requirement specifications are impossible” where scientists are concerned, and that requirements rather “emerge” on an ongoing basis. The suggestion is that this is a distinctive feature of scientific programming, which makes the application of software engineering techniques more difficult.

Verma et al. (2013) and Umarji et al. (2009) cite tighter budget and timetable constraints as a differentiating factor of bioinformatic software development, and therefore as one possible cause of a lack of software engineering best practices in that field.

Some of these responses are addressed in Chapter 6.

A number of authors point out cultural differences between scientists and software engineers as an important issue. Segal & Morris (2008) suggest that due to the subject domain complexity already mentioned, developers are likely to be the end-user scientists. But as Verma et al. (2013) point out, “[t]he primary stakeholders are biologists rather than computer scientists.” and they “may be more inclined to sacrifice program structure to get something that works”. Similarly, Swertz & Jansen (2007) state that “[the] biologist wants to apply software tools to increase the understanding of biological function without having to ‘tinker under the hood’”. Prabhu et al. (2011) point out that “given the current outlook of scientists on software and programming in general (‘scientists are not interested in software as a first order concern’, as noted by one researcher), education by itself might not be a very effective solution.”

Naturally enough, the solutions proposed by these studies flow from the diagnoses of the problem. Those who conclude that the problem lies in education propose
improvements to curricula. Those who implicate incorrect methodologies suggest alternatives that are more suitable to bioinformatics. Papers which emphasise the disconnect (real or perceived) between scientific and software engineering worlds don’t offer suggestions about how to bring software engineering values into the scientific community, which again is natural, given their premise.

The problems in bioinformatics reviewed above can be categorised in different ways, but they boil down to two overarching problems: A difficulty in reproducing bioinformatic experimental results (reproducibility) and limitations when running experiments that require very large amounts of data and/or computation (scalability).

The consequences of these problems can be similarly summarised: The bioinformatic community arrives at findings more slowly than it otherwise might, and those findings, when arrived at, are less reliable than they might otherwise be.

To better understand the nature of these problems, we can identify more precisely how and where they arise. Why does a discipline which in principle combines the two disciplines of biology and computer science suffer from such a blind spot with regards to one of these disciplines?

2.2. A Short History of Bioinformatics

The meaning of the term bioinformatics has evolved since it was first created and continues to mutate today. It is not sufficient to present one single definition, so what follows is instead a short history of the term. Disputes from the relatively recent past over the meaning of bioinformatics, or the role of a bioinformatician, can shed some light on the problems being experienced today, and indicate a direction for the future.
2.2.1. Definitions of Bioinformatics

Paulien Hogeweg (Hogeweg & Hesper 1978) is credited with first coining the term in a 1978 paper, but as she has since pointed out, she and her team had been using this word since 1970 to describe their research. The original meaning of the word was “the study of informatic processes in biotic systems” (Hogeweg 2011) and this reflected its roots in theoretical biology. This definition of bioinformatics conveyed the insight that life involved the accumulation, transmission and interpretation of information, a concept that is evident when we speak of *genetic code*.

The topics of research in bioinformatics were often a clear application of computer science to biological problems (e.g. “Spatial pattern formation ... contrasting Turing systems with gradient-based systems”) (Hogeweg 2011). This was happening in the context of a bi-directional flow of ideas between the computer scientist and the biologist, each taking inspiration from the other’s field. In one direction, Artificial Neural Networks (ANNs), genetic algorithms and actor-based systems all arose from biological analogues. In the opposite direction, it was felt that “the re-introduction of biologically inspired computational ideas back into biology was needed in order to begin to understand biological systems as information processing systems” (Hogeweg 2011).

From the late 1980s, however, “bioinformatics” was used more and more to refer to “the development and use of computational methods for data management and data analysis of sequence data” (Hogeweg 2011). This definition has become so accepted that the late 1990’s has been referred to as the “infancy” of bioinformatics (Ouzounis et al. 2012), despite its much longer history: For the general public, and even among biologists, this appeared to be a new field. Graduate courses in bioinformatics were being created. The beginning of the exponential explosion of data which continues to this day was becoming apparent (Schatz & Langmead 2013, Sagoff 2012, Marshall 2008), and the research community answered this challenge with global databases and ontologies. Today, the Merriam-Webster definition echoes
this change by describing bioinformatics as “the collection, classification, storage, and analysis of biochemical and biological information using computers, especially as applied in molecular genetics and genomics”.

This definition is looser compared to its original meaning. The more specialised - and original - meaning might today be rendered as Computational Biology.

2.2.2. Biologists, Bioinformaticians and Computational Biologists

If there is some doubt about the definition of bioinformatics, there is confusion and even controversy in the literature about what constitutes a practitioner in the field. The first bioinformaticians were quite simply biologists who had intuited the presence of informatic processes at the heart of living systems (Hogeweg 2011). In the absence of formal qualifications in the new field, it was populated by those biologists who shared those intuitions and interests.

By the time we get to 2003, a division between biologists and informaticians had begun to emerge. The bioinformatics field saw the arrival of “computer geeks who had come ... for the express purpose of getting in on the hot new thing” (Stein 2008), prompting Lincoln Stein, a biologist, to predict that by 2012 there would be no such things as bioinformaticians “as a discipline separate from mainstream biologists”. Stein updated his views in 2008, stating that “biologists are all bioinformaticians now” (Stein 2008). Though this should in a sense dilute the need for the specialised term bioinformatician, the growth of bioinformatics as a named discipline has continued, strengthening the Merriam-Webster definition cited above.

In 2017, Markowetz (2017) echoed Stein’s sentiment but from the opposite side of the barricade. As a mathematician turned computational biologist, he argued that “all biology is computational biology”. Markowetz’s opinion was in opposition to a continued scepticism on the part of “old-school” biologists with regards to computational research - one editor-in-chief had gone so far as to “use the term
‘research parasites’ to describe computational biologists making sense of published data” (Markowetz 2017).

At the core of the violent agreement between biologists and informaticians is the assertion by both sides that the computational aspect of the field is more than just a “service” to the “real” research being done by biologists. Rather, “computational thinking and computational methods are .. central to the quest of understanding life” (Markowetz 2017).

If there is a fracture with regards to bioinformatics, it is a compound one: Wet lab biologists can see computational biologists as secondary and subservient to real biology - becoming “research parasites” when they represent themselves as a separate discipline. Computational biologists, meanwhile, regard those bringing software engineering perspectives and skillsets to biological questions as “computer geeks” looking to employ their “pet techniques” (Lewis & Bartlett 2013), devoid of biological knowledge and “having to ask trivial questions constantly” (Lewis & Bartlett 2013).

When we look closely at these divisions, the fracture lines appear not between those who consider computation central and those who consider it peripheral, but between those who identify themselves as researchers in a discipline, and relegate others to the role of service providers.

A parallel can be drawn between this division, and the evolution in the meaning of the term bioinformatics, as outlined above. In Cambridge in 1952, Alan Hodgkin and Andrew Huxley researched the electrical excitation of neurons and published their findings in terms of a set of differential equations (Hodgkin & Huxley 1952). This was a clear example of the “informatic processes in biotic systems” definition coined by Hogeweg - even if it predated that definition by almost 30 years - as it identified the computational nature of a biological process.

Contrast this with a 1981 paper from Temple Smith and Micheal Waterman (Smith & Waterman 1981), which describes an algorithm to compare two sequences
(e.g. of bases or amino acids) and calculate a similarity score. The Smith-Waterman algorithm was not presented as a natural phenomenon of say Deoxyribonucleic acid, but as a useful tool to be put in the hands of those researching Deoxyribonucleic acid. Their work fits easily into the later definition of bioinformatics: “the development and use of computational methods”. Note that while Hodgkins and Huxley were both physiologists, Smith was a biomedical engineer and Waterman was a computer scientist.

These contrasting landmark publications show that some aspects of computation within biology are integral to the discipline of biology, while others are “merely” of service to it.

A failure to make such a distinction contributes to the tensions between service and integrality within the bioinformatics field described above. As mentioned above, the more recent view of bioinformatics emphasises data analysis and data management, which leans towards the service view of software. Bioinformatics researchers are more likely to want to use software than to create it (though very often they are still obliged to create it).

2.3. Current Issues in Bioinformatics

2.3.1. Reproducibility in Bioinformatics

“We do not take even our own observations quite seriously, or accept them as scientific observations, until we have repeated and tested them” - Karl Popper, The Logic of Scientific Discovery (Popper 1935).

The importance of reproducibility in experimental design is well established in the scientific community and has been addressed by philosophers of science, including Karl Popper, quoted above. In that same book, Popper suggested that a true discovery may indeed be defined as “that which can be regularly reproduced”. The concept is central to progress in the sciences because it touches on the growth of
knowledge (which can be seen as the aim of science itself). If new knowledge is founded on old, those foundations must be solid.

Goodman et al. (2016) propose more precise language around the topic. They note that words like reproducibility, replicability, repeatability etc. are often used interchangeably. When these different words are taken to have distinct meanings, it is not with any consistency. Their analysis refocuses on the idea of truth as the end goal, rather than reproducibility itself. They propose limiting the latter term to an operational sense and applying it to three different contexts:

- Methods reproducibility
- Results reproducibility
- Inferential reproducibility

Much attention is currently being given to a so-called crisis in reproducibility in scientific research generally, and, with focus on specific kinds of reproducibility, in bioinformatics. Warnings about reproducibility have been conveyed in a number of widely-cited papers.

In a 2012 Nature article, Glenn Begley and Lee Ellis reported that, in trying to confirm the findings of fifty-three landmark biomedical papers, they were successful in only 6 (11%) (Begley & Ellis 2012).

Begley and Ellis' findings were in relation to preclinical cancer research, where they found multiple issues, including “difficulties of mimicking the human micro-environment in preclinical research”, and a “lack of robust supportive data”.

The sense of a crisis in scientific research, and in reproducibility in particular, has steadily risen over the last few decades (Goodman et al. 2016). According to a Nature survey by Monya Baker (Baker 2016), 90% of researchers, when asked if there was a reproducibility crisis, responded in the affirmative, with more than 50% believing the crisis to be “significant”. Those surveyed were asked to select the
factors contributing to the problem. Lack of availability of methods, code or data were implicated at least part of the time, by around 80% of the respondents.

In bioinformatics papers, issues of methodological reproducibility can touch on matters of how code and data are recorded and preserved. One of the most infamous recent cases of failure to reproduce findings is that of Potti et al. (2006), ultimately retracted in 2011 in large part due to problems encountered by Baggerly et al. (2009) in trying to reproduce the findings.

The Potti et al. (2006) paper was an example of “high throughput” biology, where many questions (e.g. is a mutation present? is a gene switched on?) can be asked at the same time, thanks to automation, technologies such as microarrays, and computational analysis.

The paper claimed that by combining data about the effects of chemotherapeutic drugs on a standard set of human tumour cell lines (called NCI60), with gene expression signatures for that cell line (a picture of which genes were switched on and off, according to a microarray study, for those cells), a prediction could be made about the effectiveness of a particular drug based on the genes expressed by the targeted tumour cell line. The paper generated a great deal of excitement as it offered a direct route to clinical application, an idea made clear by the title of the paper: “Genomic signatures to guide the use of chemotherapeutics”.

Baggerly et al. (2009) found that the data processing aspect of this work was not described well enough to allow for reproduction, although the raw data itself was available. Thus they were obliged to employ a kind of “forensic bioinformatics” to figure out the presumed methods by working backwards from the results to the raw data. The missing documentation not only made reproducibility difficult, it masked the errors that they eventually found - errors that were sometimes simple in nature (e.g inversely labelled results) but devastating in their consequences. By the time that their findings were published, clinical trials based on the unreproducible findings and led by the main author, had already been allowed to start.
When the criticisms of the Potti et al. (2006) paper were finally and generally accepted, not only was it retracted, but the author, the head of the lab, and Duke University were sued by patients, and the families of deceased patients, involved in those clinical trials.

Peng (2015) suggests that in the end, reproducibility was not the problem in this scandal but instead, poor underlying statistical understanding and practices were to blame. But in a parallel to the legal maxim that “justice delayed is justice denied” the difficulty in reproducing the results, and thereby uncovering the underlying errors in labelling and statistics, resulted in a 5-year gap between publication and retraction in this case. In the meantime, countless hours and research dollars were lost in following this false lead, not to mention the consequences for patients in clinical trials where this research directed therapeutic decisions. Indeed as Peng (2015) himself notes in the same paper, a major component of a reproducible study is that “the statistical code and documentation to reproduce the analysis are . . . available”.

Of note in all this, is the comment by Baggerly et al. (2009) that given “the most common errors are simple”, conversely “the most simple errors are common”. This observation gives concrete motivation for the widespread sense of crisis that Baker (2016) has reported.

The case of Potti et al. (2006) is unique in the extreme nature of the fallout but not in the nature of the underlying problems.

When Ioannidis et al. (2009) investigated reproducibility in microarray gene expression analyses they concluded that where problems existed, they were primarily due to the unavailability of the original raw data and to a lesser extent an unavailability of the software used and a lack of clarity as to the methodology employed.

If the problems are well known, can the same thing be said of the solutions?

Baggerly et al. (2009) make a simple suggestion to improve reproducibility: Deliver the code with the data.
Peng (2011) lists a number of barriers to reproducibility that are specific to the computational context. Examples of such barriers include “computer code ... no longer available”, the “lack of an integrated infrastructure for distributing reproducible research to others” and the “lack of a deeply ingrained culture that simply requires reproducibility for all scientific claims”. The solutions to address this list of issues have been the subject of software engineering research and practice for decades, as outlined in section 2.4.

Sandve et al. (2013) draw attention to some of the particular difficulties in reproducing computational research, citing “new tools and technologies, massive amounts of data, interdisciplinary approaches, and the complexity of the questions being asked”. They enumerate “Ten Simple Rules” to follow in order to address these difficulties, and some of these rules lean on the techniques which have been developed and used in software engineering as described earlier, with regard to reproducibility of code and reproducibility of build. However the authors underestimate the potential sophistication of source control systems (in terms of management and use), suggest archiving versions of programs (rather than leveraging the more efficient dependency-management features of build systems) and don’t mention potential problems in reproducing execution. From a software engineering standpoint, their solutions would be considered a good start, but naïve in the context of medium-to-large development projects.

Tan et al. (2010) identify concerns about “disappearing databases, lack of interoperability ... and general quality and integrity issues”, and point to, among other things, “infrastructural and informational interoperability, such as use of international computational grids and cloud computing as backend computing resources” to address these concerns.

Kanwal et al. (2017) assembled a genomic workflow case study for the purposes of firstly identifying assumptions implicit in three different approaches to workflow (“considered needless to be stated”), and secondly making recommendations to mit-
igate such assumptions. One identified assumption was that “[t]he computing platform is preconfigured with the base software required by the workflow specification”, and the related recommendation was that experimenters should “ensure compatibility of the computing platform deployed by a researcher to reproduce the original analysis”. Similarly, they identified the “[a]vailability of specific tool versions and setting relevant parameter space” as a basic assumption. Accordingly, they recommended that such tools be “packaged along with the workflow or made available via public repositories”. These issues and their related recommendations can be seen to have parallels in the literature around software engineering reproducibility presented in section 2.4.2, especially with regard to reproducibility of deployment and execution.

Lewis et al. (2016) use a similar approach to Kanwal et al. (2017) - a set of case studies to elucidate issues and solutions around reproducibility. They emphasise a difference between replicability and reproducibility, where the former involves getting the same results, whereas the latter requires arriving at the same conclusions. This is an example of the variety of meanings which the term “reproducibility” is assigned. In this case, the definition of “reproducibility” from Lewis et al. (2016) fits into the category “inferential reproducibility” in the schema proposed by Goodman above, whereas “replicability” corresponds to “results reproducibility”. Lewis et al. (2016) make a similar recommendation to Kanwal et al. (2017) with regard to software engineering’s reproducibility of execution, and take it further by recommending Docker specifically as a suitable tool: “Provide practical, comprehensive advice on installation. Check it by installing software on commonly-used systems, or simplify it using a platform such as Docker” (Lewis et al. 2016).

Some common themes can be discerned from the work of Peng (2015), Sandve et al. (2013), Ioannidis et al. (2009), Tan et al. (2010), Kanwal et al. (2017) and Lewis et al. (2016).

- Firstly, they all acknowledge the importance of reproducibility in research.
• Secondly, they all recognise that there is a problem within computational science (including bioinformatics) in achieving this reproducibility in experimentation and publication.

• Thirdly, they agree on the nature of those problems - limitations or failure to reproduce code, build and/or execution of software used to arrive at scientific findings.

• Fourthly, where they present solutions, those solutions tend to be larger or smaller subsets of the solutions that software engineering research and practice has developed over recent decades.

However in most cases, there is a tacit assumption that such solutions can be implemented by researchers themselves, and that the skills and experience required to do so can be acquired alongside their core research activities.

As outlined in section 2.4.1 the sophistication of software engineering solutions for reproducibility has grown and continues to grow, to the point that it is not easily mastered by non-specialists. What is true of reproducibility in this regard is also true in the case of another challenge to bioinformatics identified in this thesis - scalability.

### 2.3.2. Scalability in “Big Data” Bioinformatics

In the context of software systems, though the word “scalability” is used a great deal in academic literature and in professional discourse, it is not always clearly or fully defined. Hill (1990) exhorts us to either “rigorously define scalability” or simply to “stop using it to describe systems”. Seventeen years later, Duboc et al. (2007) considered this view still valid, but went on to propose a framework to characterise and analyse software system scalability. The main point was that scalability is a “multi-criteria optimisation problem” (Duboc et al. 2007).
The paper quotes a number of authors to provide different examples of what can be meant by scalability. One of these is Brataas & Hughes (2004), who suggest that a system is scalable if it reacts “with a linear (or sub-linear) increase in physical resource usage as capacity increases”.

It is a version of this latter definition of scalability that will, with apologies to Hill (1990), be used in this thesis. It communicates only one criterion of the multiple example criteria cited by Duboc et al. (2007) - that of resource usage (and consequentially of performance) - but this is a widely-used definition and is central to scalability in bioinformatics, as we will see.

The term big data has entered the lexicon of both software engineers and bioinformaticians over the last 2 decades. As outlined by Sagiroglu & Sinanc (2013) it is typically evaluated along three dimensions: Volume, Velocity and Variability.

- Volume refers to the amount of data, moving from Terabytes to Exabytes and beyond.
- Velocity refers to the speed at which data (or changes to data) enter a system, moving from batch to stream processing.
- Variability refers to the differences in structure of that data, moving from structured, through semi-structured to unstructured.

Even if we limit the discussion to the best-known reference databases (for example the International Nucleotide Sequence Database Collaboration (INSDC)), ignoring the data generated every day by individual labs, the volumes of data are very large and growing exponentially. GenBank, which is a part of INSDC, holds approximately 200 million sequences (more than 200 billion bases). In words that echo Moore’s Law, their website\(^a\) points out that “[f]rom 1982 to the present, the number of bases in GenBank has doubled approximately every 18 months.”

One reason for this growth has to do with the falling costs of sequencing (see figure 2.1). This is indeed big data with respect to volume, and if the exponential rates of growth are maintained, it will become high-velocity within a decade or so. As many authors have warned (Schatz & Langmead 2013, Sagoff 2012, Marshall 2008), the bioinformatic community is producing data more quickly than it can analyse them or even store them. This problem is not a biological one - it is an economic one. For as long as it is cheaper to produce data than to analyse it, there will always be a bioinformatic bottleneck. As such, it requires an economic response and one such response is to be found in the use of existing, industry-proven, engineering solutions to big data problems.

The low-level techniques for scalability through parallelization, described in section 2.4.2, have been widely employed in bioinformatics (Tarmyshov & Müller-Platthe 2005, Stamatakis et al. 2005, Stamatakis 2006).

Some software engineering systems designed to tackle big data problems have been applied in the bioinformatics sphere. The Hadoop Distributed File System
Figure 2.2: Hadoop Distributed File System architecture

Shows data blocks, data nodes, name node and rack awareness.

(HDFS)\(^b\) and its associated map-reduce approach to data processing, together simply known as Hadoop, is one such system. Taylor (2010) catalogues a number of bioinformatic algorithms that employ Hadoop, including Crossbow (Langmead et al. 2009), Contrail\(^c\), Myrna (Langmead et al. 2010), CloudBLAST (Matsunaga et al. 2008) to name just a few. In a paper from the Cork Institute of Technology (CIT), O’Driscoll et al. (2015) - implemented HBLAST a Hadoop-based version of Basic Local Alignment Search Tool (BLAST).

The HDFS approach, as outlined in figure 2.2, is to explicitly represent the distributed, rack-based nature of data on modern commodity hardware. Data is stored in large blocks, distributed over nodes within racks. A node is chosen at random by HDFS to be the Name Node and process requests to store and retrieve files on the other (Data) Nodes.

Hadoop supports processing of that data by means of the MapReduce technique. MapReduce (Dean & Ghemawat 2008) is a “programming model and an associated implementation for processing and generating large data sets”. Figure 2.3 outlines how MapReduce works, using an example of calculating word frequency in a large, distributed file.

\(^b\)http://hadoop.apache.org/
However MapReduce is a very specific and even narrow programming model which is suitable for some applications but by no means all. Reshaping bioinformatic questions in such a way as to be answered by the MapReduce programming model can be difficult and ultimately increase the cognitive load on developers rather than reduce it. In addition, Hadoop was designed to work with files rather than with data in memory. Data transformations that require multiple steps must store each step on disk. As a result, Hadoop is generally considered suitable only in batch processing mode rather than real-time, due to the resulting high latency.

Spark, the opensource product based on Zaharia’s Resilient Distributed Dataset (RDD) data structure mentioned in section 2.4.2, has been presented as an easier-to-use and more performant alternative to Apache Hadoop in analysis of next-gen sequencing analysis (Wiewiórka et al. 2014). As well as the RDD’s more intuitive way to describe distributed and parallel computations, Spark performs data transformation on RDDs *in memory*, overcoming the high-latency issue in Hadoop and making Spark suitable for real-time processing. Figure 2.4 gives an overview of how Spark performs data transformations on distributed data.

The reason for the lag in moving to higher-level design-centric techniques of par-

---

**Figure 2.3.: MapReduce word frequency example**

Very large text file is distributed over multiple nodes in ‘partitions’.

Map function runs in parallel, local to a data partition, and for each word in the partition, creates a key value pair: the word, and a count of ‘1’.

Reduce function collects the output of all Map functions and combines all keys of the same value in some way - in this case by summing them, giving the word frequency.
allelization in bioinformatics from low-level programming-centric approaches is not clear, but we can reasonably speculate that it is due to a culture of programming and scripting amongst bioinformaticians, rather than one of software engineering.

2.4. Research in Software Engineering

Before examining software engineering research with respect to reproducibility and scalability, it is reasonable to first ask the question, “what is software engineering?” We can address the question to a large extent by reference to the engineering discipline in a wider sense. Koen (1984) describes the engineering method as “the strategy for causing the best change in a poorly understood or uncertain situation, within the available resources”. This characterisation of engineering as bringing about change, in particular in a context of limited resources which must be traded off against each other, is sufficiently wide to encompass the various sub-disciplines - including software engineering. The strategy referred to by Koen, is built on heuristics, which he defines as “anything that provides a plausible aid or direction in the solution of a problem but is, in the final analysis unjustified, incapable of justification, and fallible.” This emphasis on the tentative nature of heuristics is a good fit for the practice of modern software development, which is iterative and experimental.
in nature. Building on Koen’s definition, software engineering can be said to be the use of heuristics to bring about change in software systems, in a context of finite resources and uncertainty.

2.4.1. Reproducibility in Software Engineering

Current practices in commercial software engineering are the result of decades of research in the field of Software Configuration Management (SCM) (Estublier et al. 2005). This research has included various definitions of SCM and decompositions of its component parts (Dart 1991). Estublier et al. (2005) define the objective of SCM thus: “to ensure a systematic and traceable software development process in which all changes are precisely managed, so that a software system is always in a well-defined state”. The same paper calls out the key elements of SCM:

- **Versioning** in order to “maintain a historical archive of a set of artifacts as they undergo a series of changes”.

- **Building** to facilitate “[d]eriving an executable program from a set of source files”.

These techniques for achieving “well defined states” can be restated in terms of outcomes related to reproducibility:

- **Versioning**: Reproducibility of source code

- **Building**: Reproducibility of executable

In light of recent innovations around containerization (Turnbull 2014), we can add a third desired outcome: Reproducibility of deployment/execution.

Reproducibility of Code

A project’s source code is both an output and an input. It is the output of the engineering effort - the repository for all design and coding decisions made during
the project. But it is the input for the build process, which converts a static codebase into a dynamic executable. To reproduce executable software, it is essential to be able to reproduce that executable’s source code. That is, to replicate the files of a project and their relative structure, and ideally to do so for any point in its history.

SCM, of which source control is an integral part (Paulk 2002), is “now unanimously considered to be essential to the success of any software development project” (Estublier et al. 2005).

**Reproducibility of Build**

Estublier et al. (2005) identify the Make program (Feldman 1979) as “the classic tool for building”, adding that “Make (and Make-like) tools are still the most widely used tool for system building”. Since that 2005 paper, the situation has changed significantly. Estublier et al. (2005) noted that in the Java development world, tools like Apache Ant (Moodie 2006) “create[d] a higher-level build tool that may replace Make and its siblings in the long run.” This turns out to be a prescient view, as Ant became the *de facto* build tool in Java development. Since then, Ant has itself been largely replaced by Apache Maven (Miller et al. 2010).

Creating an executable from source code can be a complex task, and a number of factors make it difficult to reproduce, even when the source code is correctly versioned. Firstly, compilers and linkers can be run with different configurations. Secondly, builds depend on external binary libraries - not contained in the source control system - that must be present and identical in version to previous builds. Both Maven and later versions of Make address these obstacles to reproducibility by declaring the rules for the build and by introducing systems of dependency management for external libraries. Later Make versions included the concept of a Bill of Materials (BOM) to describe the non-source code ingredients of a build. Maven’s Project Object Model (POM) is a declarative description of a project’s build, containing configuration settings, environment values, and a catalog of dependencies.
including their versions.

Much SCM research has been dedicated to improving build reliability, from the arrival of Make in 1979, to the modern wave of tools based on Maven. Such tools are central to the everyday activities of software engineers. Though they can be complex to master, they are considered “essential in various software development life-cycle phases like unit, integration, and system testing” (Midha 1997).

**Reproducibility of Execution and Deployment**

In 1995, Sun Microsystems introduced the Java programming language which, with its Java Virtual Machine (JVM), promised programmers that they could “write once, run anywhere” (Gosling et al. 2000). The JVM was trying to solve the problems that developers faced when trying to run their executables in varied operating systems and on diverse hardware. The Java compiler does not create machine code for a given CPU instruction set, but instead creates bytecode which is then interpreted (or just-in-time compiled) by an underlying Java Runtime, specific to the target CPU. The JVM, which was followed by Microsoft’s equivalent, the Common Language Runtime (CLR) (Gough & Gough 2001), went a long way to providing reproducibility of runtime environments.

But Java is a minority language in scientific computing generally, coming behind C, C++ and Perl according to Nguyen-Hoan et al. (2010) and more recently confirmed by Russell et al. (2018). Without a JVM or similar abstraction in place, runtime environments can vary by operating system, OS version, installed libraries, installed programs, etc.

Even where Java is used, programs can still fall foul of other runtime dependencies such as JVM version expectations (Hassan et al. 2017), configuration file formats and encodings, and missing or incorrect drivers for peripheral and I/O devices.

As levels of software sophistication move from scripts, to compiled programs, to systems of cooperating programs distributed across multiple servers, the task of soft-
ware execution requires a non-trivial preceding step, called deployment: the placing and configuration of executable artefacts and required data on “web server, virtual machine or app store” (Adams & McIntosh 2016). The difficulty in reproducing the deployment and execution of such complex systems increases apace.

The servers onto which software is deployed can be either physical or virtual (Barham et al. 2003). Physical servers are, as the name suggests, units of computing hardware with CPU, memory, disk, and network resource, all managed by an Operating System (OS) which is installed directly on that hardware. Virtual servers or Virtual Machines are different. Rather than installing an OS directly on the hardware, server virtualization entails installing a hypervisor on the hardware, and subsequently installing one or more VMs on the hypervisor. The hypervisor creates a layer of indirection between the physical hardware and the OS on the VM, and this indirection brings a number of advantages including “improved utilisation, manageability and reliability” (Uhlig et al. 2005).

Virtualization makes it easier to exactly replicate a particular server software environment, including the OS type and version as well as the installed application software, and so is applicable to issues of reproducibility of execution. The advantages of virtualization have been recognised as relevant to research in the life sciences (Grüning et al. 2018).

Docker, a relatively recent innovation (starting in 2013) addresses these issues in reproducibility by extending but also simplifying the idea of virtualization.

Although Docker can be seen as simply the latest evolution in virtualization, it operates at a more fine-grained unit, called a container, than a VM (figure 2.5).

Docker requires no hypervisor - processes running in a container are running directly on the underlying host OS. But these container-based processes can not see or access any other processes or resources outside their own container. This isolation is achieved using an underlying Linux mechanism called Linux namespaces. Moreover, these processes are limited in their use of the underlying host OS resources,
Virtual machines require a hypervisor and multiple guest operating systems, whereas containers run directly on the host kernel by means of Linux cgroups (Merkel 2014).

This finer granularity results in a qualitative difference between the role of VMs and the role of containers in software architectures. A Docker image - the file that contains everything needed to run a Docker container (see figure 2.6) - is typically orders of magnitude smaller than the equivalent VM image file. This means that Docker images can be used as the output artifacts of build processes. The definition of such an artifact, including as it does the runtime stack for a program, becomes part of that program’s architecture. The Docker image becomes its unit of deployment.

In order to execute a Docker container, one only needs to have Docker installed on the host server. This enhances reproducibility by reducing the complexity involved in deploying other people’s work, encapsulating the entire stack of runtime dependencies into a single virtualized environment, and providing a lightweight file format that is easily shared.

**Layered immutable nature of Docker images**

The two defining properties of Docker images are their immutability, and their layered structure. Images can never be updated. Instead, a copy-on-write mechanism
Dockerfiles are text recipes to create binary Docker images, which are then run as Docker containers.

Images are composed of immutable layers, each upper layer potentially overriding data in lower layers. This is employed where changes to files are expressed through the creation of new instances of those files in a new layer on top of the previous one. Figure 2.7 shows the relationship between Dockerfile instructions and the layer which each creates. Each layer is an image in its own right, containing a reference to the image below it, and any changes (files that have been added, deleted or edited) with respect to that underlying image.
An image, then, is simply the final layer in a series of layers going back to a Base Image. Each layer/image is named using a hash of its contents. A change to the contents would mean a change to its name. In this way, users of the image are guaranteed that the contents are completely reproducible.

Immutability enhances reproducibility by ensuring that runtime environments based on a given image will always have the same content. The layered structure of images enhances reproducibility by making the sharing of these files more efficient and practical.

If Java artifacts virtualize a program with all its dependencies, Docker artifacts (called images) “virtualize at the operating system level” (Merkel 2014).

The advantages of Docker have been recognised in software engineering research (Cito et al. 2016) and are now finding currency in bioinformatic settings (da Veiga Leprevost et al. 2017).

2.4.2. Scalability in Software Engineering

In 1965, Gordon Moore\textsuperscript{d} made a prediction. He wrote that the density of microelectronic components on an Integrated Circuit (IC) would roughly double every year (Moore 2000). The accuracy of that prediction, later established as Moore’s Law, has become legendary and even led to speculation that it was a self-fulfilling one (Schaller 1997). Whether or not such speculation is well-founded, Moore’s Law has almost certainly set expectations on the part of the microelectronics industry, the software community, and even the public at large, with respect to the rate at which new computers should be speeding up over time.

Software is typically written to run in a serial fashion. Problems are decomposed into smaller problems, until they can finally be expressed as a set of coding instructions. Those instructions are executed serially by a processor.

But an alternative approach is possible. In this alternative, a problem is broken

\textsuperscript{d}Co-founder of Intel
down into discrete parts, some of which can be solved in parallel (Barney et al. 2010). That is to say, the resulting instructions can be run at the same time on separate processor cores (either on a single computer or across multiple computers). Compilers can automatically convert some serial code into parallel processor instructions, for example by recognising opportunities to *vectorize* data, or to convert loops into multiple threads. But such opportunities and their effects are limited, when compared to explicitly parallel programming techniques.

There are both costs and benefits when developing parallel software. The principal cost is the additional engineering complexity involved in developing software that can run in parallel. Existing research around the nature of that complexity, and techniques to minimise it, are examined later in this section.

The principal benefit of parallel software is the ability to speed up - to process more data in less time. In other words, to scale, as per the definition provided at the start of this section. As we will see, when it comes to bioinformatics, the ability to scale is of great importance. How can we quantify these benefits?

Two years after Moore’s law, Gene Amdahl formulated an equation to try to answer that question. He calculated the maximum speedup achievable through parallelization for a problem of fixed size (Amdahl 1967).

\[
\text{Speedup} = \frac{1}{(1 - p) + \frac{p}{s}}
\]

\[p\] – fraction of original execution time that can be parallelised

\[s\] – speedup factor of part that can be parallelised,

The logic of Amdahl’s Law is relentless. For a problem of fixed size, regardless of the speedup factor at play for the parallelisable part, the overall speedup will be limited by the portion of the problem that cannot be made parallel. As figure 2.8 shows, this seems to present a low and unbreakable ceiling for anyone looking to
improve performance through parallel programming.

Since the mid-1960’s then, until the start of the 2000’s, the prevailing wisdom and practice has been that parallelization was of only limited value, and that the most effective way to speed up software was to rely on the next wave of faster processors.

But things have begun to change. After more than 50 years, and notwithstanding assertions to the contrary, Moore’s Law is hitting hard physical limits (Track et al. 2017). One sign of this is the advent of the multi-core processor. Instead of trying to make processors faster, chip designers are aiming to make them more parallel, by containing multiple independent processing cores. This means that even software running on a single server, with a single processor, must be designed with parallelization in mind in order to make full use of the available processing power.
Despite Amdahl’s apparently unassailable mathematics, it is open to one point of attack - the assumption of a fixed problem size. As Gustafson later showed (Gustafson 1988), when the problem size is allowed to grow and execution time is kept fixed, the speedup achieved by parallelization can be linear with the increase in the number of processors (see figure 2.9).

The phenomenon of big data (Gandomi & Haider 2015) is a good example of a Gustafson problem of increasing size. Now that single-threaded code is no longer speeding up in line with Moore’s Law, the demands placed on software systems by big data have lead to innovations in parallel and distributed programming (and other techniques).

These innovations have been made both at the computer architecture level, and at the software application level. Flynn’s taxonomy (Flynn 1972) describes 4 possible computer architecture categories which support parallelization (see table 2.1).
<table>
<thead>
<tr>
<th>Flynn Category</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>SISD (Single Instruction Single Data)</td>
<td>Single CPU von Neumann machine</td>
</tr>
<tr>
<td>MISD (Multiple Instruction Single Data)</td>
<td>Pipeline computer architectures</td>
</tr>
<tr>
<td>SIMD (Single Instruction Multiple Data)</td>
<td>Vector processors, GPUs</td>
</tr>
<tr>
<td>MIMD (Multiple Instruction Multiple Data)</td>
<td>Multiprocessors/multiple computers</td>
</tr>
</tbody>
</table>

Table 2.1.: Flynn’s taxonomy of computer architectures

Referring to this taxonomy, we can identify MIMD as the architecture used in “commodity” multicore hardware - replacing the previous SISM of single core machines - as well as distributed computing using commodity machines (e.g. “the cloud”).

Much of the literature about scalability through parallelization is concerned either with High Performance Computing (HPC) - typically supercomputers with Infini-band networking - or with low-level approaches on commodity hardware to achieving multi-threaded and multi-processor execution like Message Passing Interface (MPI), Open Multi-Processing (OpenMP) or Complete Unified Device Architecture (CUDA) (Liu et al. 2004, Balaji et al. 2009, Dagum & Menon 1998, Cappello & Etiemble 2000), or indeed a combination of the three.

Bioinformaticians typically use low levels of abstraction as exemplified by OpenMP and MPI (Tarmyshov & Müller-Plathe 2005, Stamatakis et al. 2005, Stamatakis 2006). However such abstractions are widely acknowledged as time-consuming and error prone (Nanz et al. 2013).

What these two systems have in common is the use of synchronisation primitives
to protect blocks of shared data from concurrent access. The problem with this approach is that it sharply increases the complexity and cost of code (Bridges et al. 2007), obfuscates the underlying algorithms (Krieger et al. 2010) and introduces an entire class of new problems like race conditions and deadlock/livelock, which are difficult to find and fix. The overuse of locks can furthermore end up serialising code that might otherwise be run in parallel (Hong et al. 2012).

Notwithstanding the emphasis in research literature on low-level parallelization techniques, Asanovic et al. (2006), in a wide-ranging paper based on a cross-discipline study of the parallel programming landscape in 2006, point out (with reference to Hochstein et al. (2005)) the insufficiency of approaches to parallel programming that do not take into account human psychology, and in particular the cognitive load on developers who have to explicitly manage that parallelization. As Asanovic et al. (2006) point out, “[w]hile maximizing the raw performance/power of future multicores is important, the real key to their success is the programmer’s ability to harvest that performance”. In addition, low-level optimisations for performance can lead to issues with overall software design. Gürsoy & Kale (2004) point out that such optimisations “force programmers to sacrifice modularity”, which is an extremely important component of good design.

More recent papers revive older techniques in service of parallelization such as functional programming (Hinsen 2009) and actor-based programming (Haller & Odersky 2009).

Functional Programming

Functional Programming (FP) has a long history. According to Hudak (1989), FP is deeply influenced by Church’s lambda calculus (Frink Jr 1944), to the extent of being “usually regarded as the first functional language”. Functional languages, or the functional programming approach, distinguish themselves from imperative languages or approaches by having, as the “underlying model of computation”, the
mathematical function - in which state is carried around explicitly from function to function - as opposed to being composed of sequenced operations that act on global (implicit) state.

In a highly regarded lecture, delivered on his receipt of the Turing Award, Backus (1978) argued for an alternative to the conventional “von Neumann languages” of his time (languages that stemmed from the standard computer architecture of the von Neumann machine). Backus’ criticisms of those languages was mainly based around some of their key limitations, including an inability to compose elements of a program for reuse, and in general the close coupling they had to the underlying hardware (i.e. variables as memory cells, assignment statements as memory fetches, control statements as jumps). He finds fault with such languages for the “obstacles they present to reasoning about programs”.

Around the same time that Hudak (1989), working out of Yale, produced his comprehensive paper on the evolution and applications of FP -as part of his work on the Haskell functional programming language (Hudak et al. 1992) - his co-author John Hughes in Glasgow was also making the case for FP (Hughes 1989). Hughes cites the special features of functional programs: that variables in functional programs cannot change value, that functions will always return the same value given the same inputs and so can be called in any order, and therefore that expressions could be evaluated at any time. He expressed the advantages of these features purely in terms of the enhanced modularity and composition, much like Backus and Hudak had done.

But these same features also make it easier for compilers to break down programs into parallel and/or distributed components, as Peyton Jones (1989) demonstrated. When, in 2005, the Computing Research Association outlined computational challenges for 2020, one of these challenges was named “Popular Parallel Programming” (Irwin & Shen 2005). Although most languages used today in both commercial and research settings are very much the von Neumann languages that Backus bemoaned,
in today’s post-Moore’s law context, the topic of functional programming is enjoy-
ing a revival, and the choice of FP languages available to software engineering and
bioinformatics is on the rise (Hu et al. 2015).

This topic is taken up in more detail in chapter 4.

**Actor Architecture**

In a seminal paper, Hewitt et al. (1973) first proposed the *actor* as a universal model
of computation. Over the course of the intervening years, the model has been refined
2009).

An actor is governed by some fundamental principles (Agha et al. 1997):

- Systems of actors are driven by events, which consist of an actor receiving a
  message

- Actors have *acquaintances* - other actors which they know about and to which
  they can send messages.

- Actors can increase their number of acquaintances by creating new ones, or
  by coming to know of them through messages.

- Actors receive messages one at a time.

- Actors have state which is only changed by sending a message to the actor
  requesting that it change its own state.

As described by Karmani et al. (2009), the actor architecture is a programming
model that is inherently parallel. Programmers implementing the actor architecture
write synchronous code, adhering to a message-handling pattern, and some simple
rules (e.g. not mutating messages). Actor code, then, does not suffer from the
obfuscation effect of the thread-and-lock approach. As Agha et al. (1997) have
found, actor systems easily express a wide range of computational paradigms, and
provide a natural extension of programming into parallel systems. This naturalness and ease of expression means that programs designed to solve complex problems tend not to add much extra complexity of their own. Simplifying the solution domain frees the developer to reason about the problem domain. This direct relationship of the code to the problem domain also makes it easier to optimise algorithms based on knowledge of that domain.

Actor-based programming is an example of message-driven development, something which Gürsoy & Kale (2004) are careful to distinguish from the message-passing approach of MPI.

Specialised data structures that take advantage of these concepts have been developed and have generated a lot of interest - for example the RDD from Zaharia et al. (2010). These techniques have the advantage that they operate at a higher level of abstraction, describing software design and architecture, rather than low-level programming techniques. This reduces the cognitive load on developers, making it more practical to achieve parallelization over ever-increasing numbers of cores and machines.

2.5. Software Engineering Applied to Bioinformatics

Progress in scientific research in general, and bioinformatics in particular, relies increasingly on software (Prabhu et al. 2011, Wilson et al. 2014). But how well are the skills of software development embedded into the scientific community? To what extent are the typical developers of scientific software trained and experienced in software engineering?

Wilson (2006b), by the mid 1990s, realised the promise of software in solving scientific problems was taking a lot longer than expected. The “overwhelming majority” of scientists were using outdated tools and processes to develop, share and test their work - or sometimes none at all. A 2010 survey by Nguyen-Hoan et al.
(2010) confirms the problem.

By 2014, Wilson still finds that “most scientists are never taught how to [build and use software] efficiently” and “lack exposure to basic software development practices such as writing maintainable code, using source control and issue trackers, code reviews, unit testing, and task automation” (Wilson et al. 2014).

Wilson, amongst others, has tried to address these deficiencies though his work on software carpentry (Wilson 2006a) - an explicit recognition of the craftsmanship required in some aspects of software development, with an emphasis on the “small-scale and immediately practical issues” of the discipline. This is to be welcomed and encouraged, in that it addresses some of the aspects of reproducibility, in particular source control and build systems.

But software carpentry does not attempt to teach the more complex software concepts that address reproducibility (such as containerization) or scalability (such as patterns for concurrent software). In reviewing the lessons learned from many years of running software carpentry course, Wilson (2014) found that success depended on, among other things, keeping workshops short and “focused on a small set of tools that let us introduce higher-level concepts without learners really noticing”. The best practice is to expose scientists to the practical outcomes of software engineering research and development, not to try to turn them into software engineers: Scientists continued to consider time spent on software as a “tax they had to pay in order to get their science done” (Wilson 2014).

In a case study of multiple large and complex projects, Carver et al. (2007) concluded, as one of nine “lessons learned”, that “Multi-disciplinary teams are important to the success of [such] projects”. That is to say, teams comprised of scientists for their domain knowledge, and computer specialists for their “technology expertise”. The same study found that Agile development processes were better accepted than “traditional methodologies” by scientific developers.

Faulk et al. (2009) points out that the “broader computing community has expe-
rienced and addressed many [reliability, portability and productivity] issues in other domains” but that a “software chasm” had opened up between software engineering and scientific programming, and that applying existing engineering solutions to scientific computing will require “far greater communication and collaboration between the software engineering and scientific computing communities”. A subsequent review of the literature (Heaton & Carver 2015) found that the “[u]se of software engineering practices could increase the correctness of scientific software and the efficiency of its development”.

Letondal & Mackay (2004) describe a particular kind of multi-disciplinary, collaborative model, developed at the Institut Pasteur, called Participatory Programming. It draws nuanced conclusions about how the roles of scientist and software developer co-penetrate, but also collaborate.

2.6. Summary

The literature reviewed here follows the journey of bioinformatics from its origins in computational biology to its current big-data, software-dependent form.

The literature describes decades of research in Software Configuration Management (SCM) and software architecture which have provided today’s software engineering practitioners with tools and techniques that ensure reproducibility of code, build and deployment. However many of these tools and techniques have not yet found purchase in bioinformatic practices. The consequences of this are also to be seen in the literature, in the form of papers that describe a “reproducibility crisis” in bioinformatics, with serious ramifications for research.

A similar pattern applies to the topic of scalability. Although much progress has been made in software research to facilitate the use of parallel techniques, and though the bioinformatics literature is similarly replete with warnings of a “data deluge”, the two threads have not yet fully met. Bioinformaticians are not familiar
with recent innovations in parallel software development.
3. Survey of Life Science Software Development

This chapter presents the findings of a comparative survey of software engineering tools and techniques employed by software professionals and life scientists. These results were published as part of Lawlor & Walsh (2015). Its findings confirm many of those reported by other authors cited in the previous chapter.

- Citations: 13
- Views: 2271
- Altmetric: 16

3.1. Survey Design

At the outset of the research for this thesis, two parallel surveys were carried out, one distributed to life scientists, and the other to developers of business software. In both cases the questions aimed to identify attitudes towards certain key “markers”
of software engineering as described in the previous section. 81 life scientists, 45 of whom developed their own software, and 36 business software developers, responded to the survey. The Likert system of questionnaire design was used, in which respondents rated their attitudes to statements from strongly disagree to strongly agree, with a total of 5 degrees to choose from. We present the results below in a form that compares the differences between the two groups. The purpose of the business software data is to act as a control for attitudes towards the software engineering “markers”. Life scientist responses are in red, business software developer responses are in blue. Note that the R language was used to prepare, format and analyse data. The survey was carried out using Survey Monkey and the data was exported in anonymised csv format.

3.2. Results

The first set of results show a distinct difference between life scientists and software engineers with regard to knowledge and use of software development tools (figure 3.1) which are designed to improve reproducibility.

It is clear that business software developers and life scientists have distinctly different attitudes towards the standard elements of software engineering infrastructure. Commercial developers almost unanimously strongly agree with the statement that build systems, source control, Integrated Development Environments (IDEs) and Continuous Integration tools are used in their place of work. Life scientists show no such consensus. The closest they come to each other is in their attitude to the statement on source control where on average they agree with it, but where a significant minority have no opinion or disagree. Source control systems are of central importance in software engineering practice, on a par with disinfectant in an operating theatre. Complete adherence to their use should be considered the norm, as

\[\text{a}\text{. The data, code and the source markdown for this paper can be examined at}\text{https://github.com/blawlor/phd-paper1.git and the code can be viewed in Appendix C.}\text{b}\text{https://www.surveymonkey.com/}\]
is borne out by the business software respondents. The other three elements should be considered similarly vital to good software engineering practice.

When it comes to processes (automated or automatable) applied using the elements of infrastructure, the distinction between life scientists and the control group of business software developers is still clear even if less pronounced (Figure 3.2). This difference is mostly a function of a reduced consensus among software engineers rather than a positive change in attitudes from the life scientists. A particular point to notice is that although there is a relatively good showing for the use of source control in the previous set of results, life scientists generally neither agree nor disagree with the use of source control branching, despite the fact that such branching is one of the main advantages of source control.

As we look at the results for practices and skills (Figure 3.3), a pattern begins to emerge. The further up the pyramid (Figure 1.1) we go, the softer the consensus among software engineers, while the attitudes of the life scientists remain more or less static. The overall picture of a clean albeit smaller separation remains.

The results dealing with goals and ambitions (Figure 3.4) present a break with the previous pattern. Rather than the software engineers falling back to the neutral
Figure 3.2.: Survey question 2 results

Comparison of development processes used by Life Scientists and Software Engineers.

position of the life scientists, the latter group shows a stronger and clearer consensus in favour of the statements presented to them. In fact there is no discernible difference in attitudes between the two camps. It is interesting that in this section we have posed our questions in a slightly different way. Rather than asking about actual use, the question is about importance. The goals and aspirations of the life scientists with regard to software architecture are no different to those of commercial software engineers. What they lack however, as indicated by the previous results, are the instruments and techniques necessary to achieve those goals.

The results of the survey confirm the deficit in bioinformatic software engineering skills, while at the same time indicating an ambition among bioinformaticians to bridge the gap. In order to address this deficit effectively, the difference between computer programming and software engineering must be taken into account. It is impractical, if not impossible, to introduce the missing software engineering expertise into bioinformatics by treating that expertise as a sub-component of bioin-
**Figure 3.3.: Survey question 3 results**

Comparison of development practices and techniques used by Life Scientists and Software Engineers.
Figure 3.4.: Survey question 4 results

Comparison of architecture and design goals of Life Scientists and Software Engineers.

informatics. Software engineering encompasses too large a body of knowledge, which is acquired by too different a form of education to simply be bolted on to existing bioinformatic curricula. Put another way, the most effective way of introducing software engineering values into bioinformatic research is to introduce software engineers themselves, by recognising the separate role of the software engineer in bioinformatic research projects, and identifying the interface between the engineer and the scientist.
4. Scalability of Computation

A version of this chapter was published as the IEEE conference paper *The Weekend Warrior: How to Build a Genomic Supercomputer in Your Spare Time Using Streams and Actors in Scala* (Lawlor & Walsh 2016). It was presented to the 2016 International IEEE Conferences on Scalable Computing and Communications in Toulouse, France. It described an alternative approach to computational scaling for bioinformatics applications, based on the actor model and using streams, all in the Scala language.

- Citations: 1
- Views: 86
- **Reference:** 2016 Intl IEEE Conference on Scalable Computing and Communications, doi 10.1109/UIC-ATC-ScalCom-CBDCom-IoP-SmartWorld.2016.0098
4.1. Background

4.1.1. Architectural Solutions to Scalability

Although there have been many premature predictions of its demise, it seems that we are now finally in a post-Moore’s Law world (Waldrop 2016). This has many implications for software development at large, as discussed by other authors (Track et al. 2017). One clear implication is that so-called general-purpose programmers will need to be able to write code that can safely run across multiple cores at the same time. At the end of 2005, the Computer Research Association produced a report entitled Revitalising Computer Architecture Research (Irwin & Shen 2005) in which they laid down a number of challenges for the industry to address by 2020. One of those challenges was entitled Popular Parallel Programming. The authors asserted that “solving this problem [had] become an absolute necessity” and that “[p]ervasive parallel programming will need languages for expressing parallelism [and] parallel programming models”.

4.1.2. The Scala Language and Functional Programming

In 2011, a team led by Martin Odersky of the École Polytechnique Fédérale de Lausanne (EPFL) won a 5-year, 2.3 million-euro grant from the European Research Council (ERC) to pursue this challenge, by further developing the Scala programming language which Odersky had invented.

Scala has a number of qualities that lend themselves to expressing parallelism, but first amongst these is the fact that it is a functional programming language. FP distinguishes itself from non-functional or imperative programming in that, rather than expressing a list of instructions that updates shared data, it describes transformations to be applied to immutable (unchangeable, read-only) data, creating new data as a result (Hinsen 2009). This immutability is key: A central cause of the complexity involved in writing large and reliable parallel systems is the need to use
locks in order to protect shared data from concurrent updates in multiple threads. If shared data is immutable then no locking is required. Functional programming was covered in greater detail in the literature review in chapter 2.

However FP is a very different style compared to imperative programming and it can be difficult to adapt to this style if one is trained and experienced in the imperative world. There is a concern that one kind of difficulty (that of multi-threaded programming) is being replaced with another (that of adapting to FP style). Scala addresses this concern by offering alternative modes of expression more similar to imperative languages, thereby reducing the barrier to retraining.

In this regard, Scala offers some advantages over other functional languages. Firstly, it runs on the JVM, which automatically means a reduction in code complexity due to the JVM’s Memory Model and Garbage Collection. Additionally, by running on the JVM, Scala enjoys access to a rich ecosystem of useful libraries and frameworks, and leverages the maturity of a runtime platform that has been under development for more than 20 years. It also makes deployment costs cheaper by running on any commodity hardware. Secondly, Scala is not only a functional language, it is also Object Oriented. This makes it more accessible to programmers familiar with imperative Object-Oriented Programming (OOP) languages such as C++ and Java, reducing the costs of retraining. Thirdly, Scala comes with a rich set of productivity tools including highly sophisticated Integrated Development Environments like Eclipse and IntelliJ.

An aspect of Scala that can make parallel programming more intuitive is the fact that its flexible syntax allows programmers to create Domain Specific Language (DSL) sub-syntaxes. This is used to good effect by the Akka library, discussed in the following section.

There are some disadvantages to using Scala. Its foot-in-both-camps approach to OOP and FP gives developers a lot of choice in how to express solutions but in a programming context, choice is not always a good thing. It can give rise
to inconsistencies of style within a program, and even conflicts between purists and pragmatists within the development community. Odersky and his team have addressed some of these concerns in later versions of the language\(^a\). Another negative aspect to Scala which could impede its uptake, is the reputation of its community for being unhelpful or arrogant with respect to new adopters, an issue addressed by Rod Johnson during a Scala conference keynote in 2013, in his capacity as a member of the board of directors of Scala’s governing body\(^b\).

More recent JVM languages such as Kotlin\(^c\) which incorporate FP styles and adds some advantages of its own (e.g. explicit nullable types) may very well provide some competition for Scala in the race to win over developers.

### 4.1.3. Akka Framework

Akka is a Scala language actor architecture implementation, and has been proposed as a useful tool for scaling bioinformatic algorithms (O’Reilly et al. 2016).

Haller & Odersky (2009) proposed a Scala-based actor implementation that ran on the JVM and bridged what they described as the “impedance mismatch” between message-based concurrency, and the thread-based concurrency of the JVM. In effect, they proposed the actor as a lightweight process abstraction, which integrated implicitly with the underlying threading model.

The Akka set of libraries (Bonér et al. 2009) is a Scala implementation of the actor pattern. The FP approach, with its associated immutability, is a natural fit to actor programming, as it enforces the rule on message immutability. Scala’s facilitation of internal DSLs comes to the fore in Akka, supporting an intuitive syntax for message handling and change of state.

Figure 4.1 shows how messages are passed in Akka. Akka actors are classes that implement an Actor interface, and provide a message-handling method to define the

---

\(^a\)https://www.scala-lang.org/blog/2018/04/19/scala-3.html

\(^b\)https://youtu.be/DBu6zmzrZ_50

\(^c\)https://kotlinlang.org/
Actors communicate with each other by sending messages to each other’s mailboxes. Those messages are processed by the actors in order of arrival. The behaviour of an actor instance is coded without any explicit reference to concurrency. When a message arrives the actor’s message handling code must not mutate the incoming message - though it may change its own state in response to that message - and as a consequence, the actor can be coded in a simple, synchronous, single-threaded way. The Akka runtime schedules the message handling onto a thread from a pool of threads.

4.1.4. Reactive Streams

The Achilles Heel in actor-based systems is the mailbox: Every actor receives messages into its own mailbox and processes these messages when it is allocated resources. This can lead to mailbox overflow if the amount of data arriving into a system is greater than that system’s capacity to process it. What is required is a simple and intuitive pattern to buffer incoming messages and signal data producers
Figure 4.2.: Thread Scheduling in Akka

The Akka runtime schedules actors’ message-handling using a thread pool.

to wait before sending any more messages. This signal is known as \textit{back-pressure} and has been implemented as part of networking protocols for decades. Generating back-pressure is a low-level activity which, similarly to the thread-and-lock pattern, will obfuscate code if implemented explicitly. Reactive Streams (Khare et al. 2015) offer a way of implementing back-pressure in a transparent way, orthogonal to the “business logic”. They work well with actor systems by avoiding the message buffer overflow problem in a succinct and elegant way.

In addition to solving the message overflow problem, reactive streams allow systems to become more robust and to scale \textit{down} as well as up. By transparently propagating back-pressure from consumers to producers of data, they allow systems to react gracefully and intelligently to limitations in processing power (either temporary or systemic) without failing.
4.1.5. Applications in Bioinformatics

The following sections 4.2 and 5.3 describe some architectural proofs of concept developed as part of the research for this thesis. They are intended to demonstrate two things. Firstly, that problems of scaling in bioinformatics can be addressed by frameworks and architectures not currently widely used by, or perhaps even known by, bioinformatic practitioners. And secondly, that the skills and experience required to recognise and apply these solutions are not part of the typical bioinformatic skillset. These solutions address two main problems in scalability - scaling computation and scaling data. The first proof of concept uses functional programming (Scala) and actor architecture (Akka) to distribute a well-known bioinformatic algorithm across multiple computational nodes. The second is a novel use of a distributed transaction log implementation (Apache Kafka) as a genomic database, distributing replicated and streamable data across arbitrarily large clusters.

4.2. Materials and Methods

4.2.1. Distributed Computation in Bioinformatics using StAcS

For the purposes of this research, a core set of tools have been selected to demonstrate the architectural solutions described above: Streams and Actors, implemented by the Akka framework, using the Scala language. For brevity, this combination will be referred to as StAcS - Streams and Actors in Scala.

The Smith-Waterman algorithm (Smith & Waterman 1981) is a well-known technique to compare one biological sequence (say DNA) with another. The output of Smith-Waterman is a score which quantifies how well two such sequences align with each other.

The starting point of the algorithm is a grid formed by the two sequences being aligned, as shown in figure 4.3. Note that if both sequences were identical, the
diagonal of the grid would coincide with the positions where the row and column have identical sequence items. The first row and columns are scored with values of zero.

\[
\begin{array}{cccccccccccc}
A & T & T & G & A & G & G & A & C & T & G & G \\
\hline
A & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
T & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
G & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
G & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
A & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
G & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
G & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
A & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
C & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
T & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
\]

Figure 4.3.: Smith-Waterman: Initial matrix

The two initial configuration values for the algorithm are:

**Scoring matrix:** Scores to apply when two sequence items are the same, or different. In this case, we’ll apply a +3 score when two sequence items are the same, and otherwise -3.

**Gap penalty:** A drop in score applied when a value in the grid is taken from a horizontal or vertical neighbour rather than a diagonal one. In this case, we’ll apply a gap penalty of 2.
With this configuration in place, we can describe the core of the algorithm. Moving through the remaining cells of the grid, from top to bottom, and left to right, score each cell with the maximum of the following four values:

- The sum of the score in the neighbouring cell above and to the left, with the score from the scoring matrix for the sequence item of the cells row and column.

- The score of the neighbouring cell to the left, minus the gap penalty.

- The score of the neighbouring cell above, minus the gap penalty

- Zero

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>T</th>
<th>T</th>
<th>G</th>
<th>A</th>
<th>G</th>
<th>G</th>
<th>A</th>
<th>C</th>
<th>T</th>
<th>G</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>T</td>
<td>0</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>G</td>
<td>0</td>
<td>4</td>
<td>3</td>
<td>7</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>G</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>6</td>
<td>4</td>
<td>8</td>
<td>7</td>
<td>5</td>
<td>3</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>A</td>
<td>0</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>4</td>
<td>9</td>
<td>7</td>
<td>5</td>
<td>10</td>
<td>8</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>G</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>7</td>
<td>12</td>
<td>10</td>
<td>8</td>
<td>7</td>
<td>5</td>
<td>9</td>
</tr>
<tr>
<td>G</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>5</td>
<td>10</td>
<td>15</td>
<td>13</td>
<td>11</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>A</td>
<td>0</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>6</td>
<td>8</td>
<td>13</td>
<td>18</td>
<td>16</td>
<td>14</td>
<td>12</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>21</td>
<td>19</td>
<td>17</td>
<td>15</td>
</tr>
<tr>
<td>T</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>9</td>
<td>14</td>
<td>19</td>
<td>24</td>
<td>22</td>
</tr>
</tbody>
</table>

Figure 4.4.: Smith-Waterman: Scoring complete
From the description of the algorithm, we will see that there is a dependency between cells when calculating values. A cell can only be filled when the three surrounding cells above and to the left have been filled. Figure 4.4 shows an example scoring.

With the grid scores complete, the cell with the highest score is located (the red cell in Figure 4.5). This is the end of the alignment. From this cell, a backtracking is performed, following the highest score backwards (up, left or diagonally up and left) until the first zero value is reached (the green cell). The sum of the scores is the overall score of the alignment.

\[\begin{array}{cccccccccccc}
A & T & T & G & A & G & G & A & C & T & G & G \\
A & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
T & 0 & 1 & 6 & 4 & 2 & 1 & 0 & 0 & 1 & 1 & 4 & 2 & 0 \\
G & 0 & 0 & 4 & 3 & 7 & 5 & 4 & 3 & 1 & 0 & 2 & 7 & 5 \\
G & 0 & 0 & 2 & 1 & 6 & 4 & 8 & 7 & 5 & 3 & 1 & 5 & 10 \\
A & 0 & 3 & 1 & 0 & 4 & 9 & 7 & 5 & 10 & 8 & 6 & 4 & 8 \\
G & 0 & 1 & 0 & 0 & 3 & 7 & 12 & 10 & 8 & 7 & 5 & 9 & 7 \\
G & 0 & 0 & 0 & 0 & 3 & 5 & 10 & 15 & 13 & 11 & 9 & 8 & 12 \\
A & 0 & 3 & 1 & 0 & 1 & 6 & 8 & 13 & 18 & 16 & 14 & 12 & 10 \\
C & 0 & 1 & 0 & 0 & 0 & 4 & 6 & 11 & 16 & 21 & 19 & 17 & 15 \\
T & 0 & 0 & 3 & 3 & 1 & 2 & 4 & 9 & 14 & 19 & 24 & 22 & 20 \\
\end{array}\]

**Figure 4.5.: Smith-Waterman: Backtracking complete**

The alignment of the two sequences can also be read off the results using the following system: Starting with the red cell, align the values from the corresponding row and column (T and T in this case). Tracking back, when moving diagonally,
simply align the two values. When moving up, mark a gap in the row sequence. When moving left, mark a gap in the column sequence.

In this case, this leads to the following alignment between the two sequences:

\[
\begin{align*}
\text{ATT - GAGGACTGG} & \quad \text{(Row sequence)} \\
| & | | | | | | | | x x \\
A - TGGAGGACT & \quad \text{(Column sequence)}
\end{align*}
\]

Smith-Waterman is known to be more sensitive but slower than rival algorithms such as BLAST. Shpaer et al. (1996) conclude that Smith-Waterman is “significantly better able to distinguish true similarities from statistical noise than ... BLAST”. In applications where it is particularly important not to miss potential matches, and to avoid spurious ones, Smith-Waterman should be favoured, notwithstanding the performance penalty incurred. There is value, then, in harnessing the power of Smith-Waterman in a scalable work-distribution mechanism.

In a typical use case, the Smith-Waterman algorithm is applied in a pair-wise fashion using one or more query sequences and a large number of database sequences. Each of these invocations is independent, and the results of all are compared at the end to choose the closest match. It is thus an example of an embarrassingly parallel problem, and amenable to linear scaling through multi-threading and distributed computing. As outlined in section 2.3.2, a drastic reduction in the cost of sequencing is leading to a commensurate rise in the amount of sequences to be aligned. But if the need to scale up and out is clear, the difficulty lies in the complexity of such implementations.

As shown in the literature review, The low levels of abstraction typically used by bioinformaticians, as exemplified by OpenMP and MPI, add complexity and are more error-prone.

Given it’s applicability to the alignment of ever-increasing amounts of genomic
sequences, the Smith-Waterman algorithm is the chosen test bed for the application of the higher abstractions of StAcS to the bioinformatics field in this research.

### 4.2.2. Proof of Concept

What follows is a high-level description of the StAcS implementation of Smith-Waterman.

The system runs across one or more network nodes (physical computers or VMs), each executing the same *worker* program. One lightweight *master* node distributes tasks to the workers and receives the results (see Figure 4.6). All nodes participate in a cluster which is managed by the Akka framework and which implements both the actor model and reactive streams.

![Diagram](image)

**Figure 4.6.: Worker nodes pulling work from Master Node**

The actors are unaware of their physical location as they communicate with each other.

The goal is to create a single actor instance per query-sequence pair, each respon-
sible for invoking the Smith-Waterman algorithm for that pair. The Akka runtime system takes care of allocating these actors to the available threads. To avoid flooding the system with too many actors, the built-in back-pressure mechanism of Akka streams is used.

Figure 4.7 shows how this works in detail:

- Every circle in this figure represents an actor instance and the arrows between them represent the creation of new actors and the sending of messages (including stream contents).

- The Alignment Worker at the bottom of the hierarchy is where the pairwise alignment of a single query sequence with a single database sequence is done.

- The Query Batch Worker at the top of the hierarchy is the point of entry to each worker node and is responsible for splitting a number of incoming queries (a batch) into individual queries to be processed in parallel. It creates a stream of Queries which are serviced by the Query Subscriber.

- Every actor level in the hierarchy between the entry point and the Alignment Worker breaks the work up further into parallel tasks; first by database (a group of known sequences) and then by individual database sequence, creating new child actors for each task.

- The speed at which this proliferation of parallel tasks is performed is managed by three types of incoming reactive streams: a stream of queries, a stream of database names and a stream of database sequences.

- The Alignment Workers, which are doing the CPU-intense task of performing the Smith-Waterman alignment, create the necessary back-pressure which is transparently propagated back up the hierarchy, throttling the rate at which the streams send queries, database names and database sequences.
Figure 4.7.: Actors on the Worker Node

Note that there is a reverse hierarchy on the right, which gathers all results and performs local comparisons of the scores, passing up only the highest score from each database, and then each query. This allows for parallelization of those comparison computations, and also dramatically reduces the number of messages exiting the worker node. The master node receives only the 'winning' database sequence for each query.

The Smith-Waterman library used was developed by Zhao et al. (2013) and is a single-threaded, highly-optimised SIMD library available also to the JVM through a Java Native Interface (JNI)\textsuperscript{d} library provided by that team.

\textsuperscript{d}Java Native Interface: a mechanism that allows code on the JVM to invoke native code
It is important to note that this design, which uses the idea of delegating increasingly fine-grained tasks to increasingly specialised actors, is intuitive both in the sense of the problem it is trying to solve (the problem domain) and in the parallelization that it employs to solve it (the solution domain). This distinguishes actor-based designs from low-level locking approaches to parallelization, which require separate and distinct design considerations for problem and solution domains.

The code itself is relatively easy to understand. Each actor does the same core task: it waits for a message and processes it. The result is a set of self-similar blocks of code, each with a small amount of logic, and each doing a small, quickly understandable task. For listings of the code behind each actor, see appendix B.

4.3. Results

4.3.1. Parallelization Over Multiple Cores

A measurement was taken of the average time to completion over 10 runs of the non-StAcS Smith-Waterman alignment as it looped through 10,000 database sequences which were read from a number of local files, on a Linux server with an Intel i7 (quad-core) processor. Similarly, the average time to completion for the StAcS implementation on the same machine, reading the same number of sequences from the same files, was measured. The results are shown in table 4.1.

As can be seen, the times taken for the alignments are consistent. The single-threaded version runs almost 5 times more slowly than the StAcS approach, demonstrating its ability to scale. The relative standard deviation across runs is no greater than 1.5% in both implementations. The significance of these results is discussed below.
<table>
<thead>
<tr>
<th>Run</th>
<th>Single Thread</th>
<th>Actor and Stream</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>219,801</td>
<td>46,972</td>
</tr>
<tr>
<td>2</td>
<td>219,434</td>
<td>45,850</td>
</tr>
<tr>
<td>3</td>
<td>223,688</td>
<td>45,653</td>
</tr>
<tr>
<td>4</td>
<td>219,401</td>
<td>45,684</td>
</tr>
<tr>
<td>5</td>
<td>227,051</td>
<td>46,702</td>
</tr>
<tr>
<td>6</td>
<td>226,643</td>
<td>45,931</td>
</tr>
<tr>
<td>7</td>
<td>224,076</td>
<td>45,995</td>
</tr>
<tr>
<td>8</td>
<td>220,702</td>
<td>45,509</td>
</tr>
<tr>
<td>9</td>
<td>225,904</td>
<td>45,709</td>
</tr>
<tr>
<td>10</td>
<td>223,557</td>
<td>45,667</td>
</tr>
</tbody>
</table>

Table 4.1.: Alignment times on single node (ms)

4.3.2. Parallelization Over Multiple Nodes

Using Digital Ocean\(^e\), a commercial cloud computing service, the worker agents described above were deployed on increasing numbers of Linux nodes and for each new node, the time to completion of a Smith-Waterman query over ten runs was measured, and the average was taken. In order to keep the experiment times to a reasonable level, rather than maintaining a fixed number of database sequences and expecting a reduced time to completion, the number of database sequences was increased by 10,000 for every new node, with the expectation of a constant time to completion. The results are shown in figure 4.8.\(^f\)

Allowing for some noise based on variable performance from individual nodes, the results show a linear increase in throughput for every extra node added. This perfect scalability should be expected given the embarrassingly parallel nature of the problem, and the StAcS approach has demonstrated its ability to meet that expectation. As will be discussed below, it meets these expectations while at the same time bringing some important advantages of its own.

\(^e\)https://www.digitalocean.com/
\(^f\)The GCUPS unit is a common way of measuring performance for Smith-Waterman and means Billion Cell Updates Per Second, where cell refers to the cell in a Smith-Waterman grid as described earlier.
Figure 4.8.: Scalability over multiple nodes

Scalability over multiple nodes remains linear as we move from 1 to 10 nodes.

4.4. Discussion

4.4.1. Parallelization Over Multiple Cores

The central finding from the results of the multiple cores experiment is that using higher abstractions to achieve multi-threading does not impair the ability to scale linearly. As noted in the review of the literature, there is a prevailing use of low-level programming as a way to achieve efficient parallelization in bioinformatics. These results demonstrate that this prevailing use is not the only option available in such contexts.

4.4.2. Parallelization Over Multiple Nodes

The purpose in showing these results is to demonstrate that the higher abstraction of StAcS does not compromise the expected scalability and instead simplifies it: The same abstractions that allowed Smith-Waterman invocations to be scaled across the cores of a single node, scaled also across the nodes of a cluster. Rather than using both OpenMP and MPI, bioinformatic computations can be distributed across local and remote processing cores using the same high-level abstraction - the actor.
4.4.3. Maintenance

The ISO 9126 model of software quality decomposes maintainability into 5 sub-characteristics:

- Analysability: how easy or difficult is it to diagnose the system for deficiencies or to identify the parts that need to be modified?

- Changeability: how easy or difficult is it to make adaptations to the system?

- Stability: how easy or difficult is it to keep the system in a consistent state during modification?

- Testability: how easy or difficult is it to test the system after modification?

- Maintainability conformance: how easy or difficult is it for the system to comply with standards or conventions regarding maintainability?

The proof of concept presented here has approximately 1000 lines of Scala code, and uses abstractions that express the parallelization problem directly in a coherent and readable way. The low line count and high readability are direct consequences of choosing high-level abstractions like actors and streams, as well as the Scala language itself which is demonstrably less verbose than C++ (Hundt 2011). These qualities have a strongly positive effect on maintainability, impacting directly on three of the four factors of ISO maintenance model: analysability, changeability and testability. Stability was not directly impacted either positively or negatively by the aforementioned qualities.

4.4.4. Further Scaling

Because the system runs on the JVM and does not rely on specialised processors, a wider choice of runtime platforms is available. This flexibility keeps the cost
of running the software low as one can “shop around” for a competitively priced platform provider.

To date, the Akka framework has been tested successfully in clusters of 2400 nodes and its development team does not believe this to be a hard limit. Assuming only quad-core processors were available on each node, this configuration would still effectively constitute a 10,000 core computation fabric, and would provide super-computer levels of performance.

Extrapolating the experiment to a 2000 node cluster - well within proven parameters of the Akka framework - the proposed architecture would yield performance of around 1500 GCUPS.

### 4.4.5. Further Work

Although actors are presented here as a high-level abstraction with respect to threads and locks, in different contexts, actors may be considered low-level abstractions on which increasingly sophisticated constructs can be built. An example of this would be the Resilient Distributed Dataset (RDD) of the Apache Spark project which uses Scala and Akka (Zaharia et al. 2012), and Akka Streams themselves which are implemented in terms of actors. While the purpose here was to advocate for the use of actors over threads, further work to investigate these higher abstractions for their applicability to bioinformatic problems would be beneficial.

### 4.4.6. Conclusions

Based on these outcomes the following can be concluded:

- The StAcS architecture has demonstrated an ability to scale up and scale out when used in a bioinformatic context. Furthermore, it doesn’t suffer from issues related to OpenMP and MPI and low-level thread-and-lock programming as outlined in the literature review.
• A parallel and massively scalable implementation of the Smith-Waterman algorithm, which runs on cheap commodity hardware in the cloud, was developed using the StAcS architecture in a modest timescale. This would lend itself to accelerated development of bioinformatics systems.

Given these points, such architectures should be considered as viable alternatives to the low-level means of parallelization described earlier.
5. Scalability of Data

This chapter is based on the paper *Field of Genes: Using Apache Kafka as a Bioinformatic Data Repository (Lawlor et al. 2018)*. It takes the work done on computational scalability from the previous paper, and develops it to address large sets of distributed data. This paper represents a coming together of the research done as part of this thesis, combining scale in computation and data, with important elements of reproducibility (covered in a separate chapter). The paper was published in GigaScience (Oxford Academic) where it has so far accumulated 5 citations, an Altmetric score of 19, and more than 3000 views.

- Citations: 5
- Views: 3028
- Altmetric: 19
- Reference: GigaScience 7(4), doi 10.1093/gigascience/giy036

A precursor to the paper was presented to the Collaborative European Research Conference (CERC) in 2016 where it won the award for Best Paper/Presentation. CERC\(^a\) is a multi-disciplinary conference that aims to bring together researchers from disparate fields in order to foster exchange and collaboration.

\(^a\)https://www.cerc-conference.eu/
The previous section describes an approach to parallelization - called StAcS - with particular emphasis on the use of CPU as Smith-Waterman is a processor-intensive algorithm. In this section, a novel solution is presented, for managing genomic data (that is, storing and providing access to that data) in a way that complements the StAcS approach to processing such data.

5.1. Background

5.1.1. The Structure of Data

Bioinformatic data is available from a number of sources. An example of this is the RefSeq database which maintains records of genomic DNA for model organisms (Pruitt et al. 2005). RefSeq is maintained by the National Center for Biotechnology Information (NCBI)^b, and its website includes two mechanisms for accessing the data: by searching for sequences using either the BLAST program, or the Entrez database system^c, or by downloading the database files used by BLAST. For an application processing this data, as opposed to just searching it, neither option is particularly attractive, for reasons presented below.

BLAST is an invaluable tool for bioinformaticians who are searching for a particular genetic sequence (Altschul et al. 1990). It performs an alignment of a query sequence against a database of known sequences, returning results based on similarity. It has speed advantages over the Smith-Waterman alignment algorithm used in the previous experiment (Shpaer et al. 1996) at the cost of reduced sensitivity. But

---

BLAST is a search algorithm, with its own specialised data structure. If we use this structure, then all we can do with the data is search it using this algorithm.

On the other hand, if we have access to the raw data, we can process it in any way we need in order to answer biological questions. NCBI provides a means to retrieve the underlying raw data by accessing the anonymous FTP server\textsuperscript{d}, navigating to the correct database folder, and downloading, unzipping and untarring the contents. In the case of RefSeq, the result is almost 1TB of data spread across a few hundred text files. While this result does indeed qualify as raw data, its localised sequential file structure limits its usefulness, in particular by hindering parallel access to the sequences it contains.

Any given dataset may be presented in many different ways for different users, depending on who they are and what they want to do with the data. The structure conditions the way in which this data may effectively be used.

One of the characteristic qualities of bioinformatic data is that it is vast and growing. From an engineering perspective, the most effective way to process this data is in a parallel fashion and so our data should be structured in a way that facilities this.

What are the requisite properties of such a structure? The following non-exhaustive list of such properties is proposed:

\textbf{5.1.2. Distribution}

In order to free the data of hardware bottlenecks like network adaptors, CPUs and memory, parallel data should be distributed across multiple machines. This is analogous to exposing the largest possible working surface area of data, in contrast to the limiting effect of storing all data on one physical server.

5.1.3. Reliability

It is a property of distributed systems to be both more and less reliable, in the sense that they no longer have a single point of failure (more reliable), but there are more elements that can break (less reliable). Distributed data’s structure should protect it from this latter effect, while promoting the former.

5.1.4. Streaming

The advantage that streaming brings is that consumers of a stream do not have to wait until all data is downloaded before beginning to process that data. An everyday example of this is Netflix - the Internet movies streaming service. In contrast with previous models of download-and-view, where the movie must be downloaded entirely before viewing can begin, with Netflix it takes 90 minutes to watch a 90-minute movie. Although not all bioinformatic processing can be performed on partial data, much of it can. That is, there can be multiple stages of processing where the output of one stage acts as the input to the next. Streaming is particularly advantageous for such data. By including this element of streaming, we take into account an extra dimension of parallelization when processing genomic data.

It should be clear that 1TB of data in flat files on a single hard drive does not present a structure that conforms to the above properties. So how can raw bioinformatic data be engineered into a structure that does?

A preliminary evaluation was performed of a number of opensource applications and frameworks created by software engineers to deal with data at scale. Redis® (an in-memory, clustered, schemaless database), Apache Cassandra® (a clustered, schema-optional, high-availability database) and Kafka (described below) were tested for their speed of read/writing when run in a cluster. All three are distributed and offer streaming interfaces, but for reasons of performance and scala-

---

*redis*: https://redis.io/
*Cassandra*: http://cassandra.apache.org/
bility, *Apache Kafka*\(^9\) was selected as the basis for a proof of concept implementation of a parallel genomic data repository.

## 5.2. Materials

### 5.2.1. Apache Kafka

The emergence of *Kafka* as the most suitable platform to proceed with was unexpected, as *Kafka* is not commonly considered to be a database. It is generally viewed as a message broker - an intermediary program that transfers messages asynchronously from one program to another via a queue or *Topic*\(^h\) - and has been identified in that role as a suitable technology for bioinformatic applications (Ta et al. 2016). However Kafka’s developers at *LinkedIn*\(^i\) implemented it with a wider scope of usage in mind, including “source-of-truth data storage” (Wang et al. 2015). Kafka’s *topics* are implemented as (Distributed) *Transaction Logs* - an abstract data type to which new data is only ever appended, never overwriting existing data. Moreover, Kafka topics can be configured to never expire, and this means that the same data can be read over and over again. These facts, combined with important speed advantages conferred by the contiguous nature of the data storage, allow Kafka to operate as a data repository with extremely high read and write speeds.

In the following paragraphs, some of the features of Kafka are described, with explanations of how they confer the desired parallel properties.

### 5.2.2. Consumers and Producers

Figure 5.1 outlines what any given Kafka installation looks like: A cluster of Kafka *brokers* on which independent *producers* (writers) and *consumers* (readers) oper-\(^g\)http://kafka.apache.org/\(^h\)A topic is analogous to a queue, but for a publish-subscribe model. Publishers write messages to a Topic and any number of Subscribers read from it.\(^i\)https://en.wikipedia.org/wiki/LinkedIn
5.2.3. Partitions

Topics are normally single entities in message brokers, but in Kafka they are divided in partitions, as shown in figure 5.2. It is these partitions that make Kafka parallel, reliable and distributed. The partitions for a given topic are spread across the nodes, distributing that topic across the cluster. The partition is also the unit of parallelization. If a topic is configured to have $N$ partitions, then $N$ consumers can read independently and in parallel - but in concert - from the same topic. Finally, topics can also be configured to have a replication factor, $R$. This is the number of copies of a partition maintained across the cluster, so that if up to $R - 1$ machines in the cluster fail, no data is lost. Incidentally, partitions also confer scalability: Topics can become arbitrarily large - holding more data than any given machine in the cluster can permit.
5.2.4. Consumer Groups

Another important feature of Kafka is the fact that it centrally manages the distribution of parallel work across multiple consumers, if these consumers belong to the same *consumer group*. The example in figure 5.3 demonstrates this by showing a small cluster with two servers and a single topic broken into four partitions. The four consumers belonging to Consumer Group B are automatically allocated one partition each, but in Consumer Group A, where there are only two consumers, each consumer is given two partitions to read.

This organisation allows a group of consumers to collaborate in order to drain a topic in parallel. Another important thing to note here is that partitions are only read by one consumer at a time, and this means that order of reading is preserved for any given partition. Because order is preserved, it can be known when a partition has been completed by adding a 'back marker' at the end. This adds another useful element to Kafka-as-a-data-store: We can do an exhaustive sweep of a topic and know when we have touched everything.
Consumers can form groups to read from a single topic by maintaining a relationship between consumers and partitions (image from Kafka website).

5.2.5. Producers and Message Keys

In contrast to the consumers which are dedicated to one or more partitions, producers can end up inserting data into any partition of a topic. That is because the decision about which partition to send a message to is driven by the message key. Every message is composed of an optional key, and a value. Producers use a partitioning strategy to select a destination partition based on the key of the message, choosing a random partition where no key is present. The default partitioning scheme calculates a numerical hash of the key and then calculates the value of the result, modulo the number of partitions. This strategy can be customised by any given producer. Assuming a well-balanced hash algorithm, this will result in messages being evenly distributed across all partitions by default.

5.2.6. Log Compaction

Another important and useful aspect of message keys is their role in log compaction. As already mentioned above, messages are continuously appended to partitions rather than overwriting old values. That said, Kafka has a mechanism for dealing with cases where new values for old messages are sent, or even where messages are deleted. This is called log compaction and works as follows: on a scheduled
basis, Kafka will recopy a partition, moving from oldest to youngest message and removing any messages who have a younger version, based on key identity. Deletion of messages is brought about when the youngest message with a given key has a null value. Note that if no keys are present, log compaction will not be available.

Thanks to this mechanism, Kafka can continue to append new messages to topics without the topic growing indefinitely. Of course where new keys are added, the amount of space needed will increase over time, and Kafka caters for this by allowing new brokers to be added to the cluster, and rebalancing the load.

5.2.7. Streaming

Finally, the Kafka API which is available in the Java language, includes support for streams. More recently, Kafka has begun to support the Reactive Streams API\(^1\), which includes the automatic management of back-pressure when chaining many streams together into a pipeline. There are many implementations of the Reactive Streams API, including a Scala-based one in the Akka framework.

5.3. Methods

5.3.1. Using Apache Kafka as a Bioinformatic Data Repository

In order to measure the performance characteristics of a Kafka-based genomic data repository with respect to flat files, and also to understand what other properties might emerge from such an implementation, a proof of concept was produced that loads up to 11% of the RefSeq database into a single Kafka topic, spread over either 8 or 12 cloud servers, depending on needs. In order to convey the image of the extended ’working surface area’ sought, this proof of concept was named Field of Genes. What follows is a description of how to build the Field of Genes, and how

\(^1\)http://www.reactive-streams.org/
to measure its performance and scalability. Measurements of its performance with respect to the use of flat files (the *de facto* alternative) are then presented.

To facilitate reproduction of our findings for independent verification, this work has made extensive use of Docker to deploy Field of Genes. The use of Docker to enhance reproducibility is discussed in detail in the previous chapter. In addition, and again in order to make this work reproducible, the full codebase and instructions for build, deployment and experimental run have been made available\(^k\).

In testing the suitability of Kafka as a bioinformatic data repository, two operations are considered:

- Loading data onto the Kafka cluster from NCBI.
- Bioinformatically processing data on the cluster.

In preparation for the experiment, some benchmark values are ascertained:

- The time taken to download, unbundle and convert RefSeq files from the NCBI FTP site.
- The time taken to run the bioinformatic process using a simple thread-and-lock technique.

With these benchmarks recorded, the experimental phase consists of three steps:

1. Preparation of the Kafka Cluster
2. Loading of the sequence data from RefSeq to Kafka topic
3. Bioinformatic processing of the sequences in Kafka topic

### 5.3.2. Preparation of Kafka Cluster

The data storage and computational fabric for Field of Genes is built on cloud VM servers using Google Cloud Platform (GCP) \(^l\). A Kubernetes cluster of hosts (8 and

\(^k\)https://github.com/blawlor/field-of-genes

\(^l\)https://cloud.google.com

101
12, depending on requirements) is created, each with 4 cores, 8 GB of memory and 80GB of SSD disk space.

On this cluster of machines, an equal number of Kafka broker instances is rolled out, one per machine. Off-the-shelf Docker images for Kafka were used, so no extra coding was required.

### 5.3.3. Loading data to Kafka from RefSeq

The StAcS architecture described in chapter 4 was used to create a **loader agent**. This is a small autonomous program which processes instructions from one Kafka topic, uses a Java library to download and convert the contents of a single RefSeq file, and then sends the downloaded sequences to another Kafka topic, each sequence getting its own record. The Java library in question was developed to download, untar, unzip and convert (to FASTA format) a single RefSeq file from NCBI. Figure 5.4 gives a representation of this agent. It shows a process which downloads from the NCBI FTP site, and pushes sequences to a RefSeq topic. But it also shows that Kafka topics are used to send instructions to the agent. When choosing how to send instructions to the agents, and how to receive responses, it made sense to use the Kafka infrastructure already in place. But this design decision had interesting effects which are examined later.
The loader agent was deployed using different levels of parallelization - 4, 8, 12, 16, 20 and 24 - to obtain an indicative download speed for each level, as well as to test for linearity of scalability. Note that as the level of parallelization was raised, the amount of data to be processed was similarly increased, by increasing the number of downloaded files. The goal was to measure how well the benchmark (described below) and the Field of Genes can adapt to high load. A scalable system should show a flat horizontal line for time taken, as both load and parallelization are increased in tandem.

Each agent was part of the same consumer group with regards to the Loader Instructions topic which, as explained above, means that the partitions of that topic were evenly allocated across the agents. The number of partitions of the Loader Instructions topic was set to the same as the level of parallelization.

The producer responsible for writing to the Loader Instructions topic used incremental numeric keys for the messages and relied on the default partitioning strategy. In this way, the instruction messages were spread evenly across the partitions.

For each level of parallelization, the elapsed time from when the first Loader Instruction message was sent, to when the last RefSeq sequence was written to its destination topic, was measured.

The benchmark for comparison was a single server of exactly the same specifications as those used by Field of Genes, using multi-threading to achieve whatever levels of parallelization the system could support. The hypothesis is that by spreading the genomic data over a wider “working surface area” levels of parallel access and scalability can be attained that more than compensate for any performance loss due to transmitting data between machines. Therefore this benchmark architecture - a single server where no network traffic is required and no streaming is performed - is appropriate.

The combined technologies of Docker and Kubernetes allowed us to reproduce the same server environment and change only the software components under test. This
was accomplished by creating a single instance of a GCP server, and deploying a single application that - with varying numbers of threads - downloads from NCBI using the same Java library the Field of Genes agents used. Note however that due to the limited disk space on a single server, it was not possible to run the benchmark to the same levels of parallelization as with the experiment. As will be seen from the results, it is still possible to arrive at a useful comparison of absolute performance and relative scalability between the benchmark and Field of Genes.

5.3.4. Bioinformatic Processing

The previous subsections described how the Field of Genes is constructed and populated with RefSeq data. In this section, the implementation of a bioinformatically useful agent that operates on that data is described.

The ratio of cytosine and guanine bases (the Cs and Gs of the genetic code) to adenine and thymine (the As and Ts), is a biologically meaningful property of any given sequence (Sueoka 1962). Measuring this value for a large number of sequences is an example of a parallel processing problem, and therefore suitable as an initial test for the Field of Genes.

The elements of this implementation are very similar to those of the loader previously described: An independent agent which consumes from some topics and produces into others is constructed and deployed using varying levels of parallelization. Figure 5.5 gives an overview of this agent.

From this we can see that the output of the Loader agent has become the input of the GC Content agent, hinting at the opportunity to parallelize these tasks into a streaming pipeline, as discussed earlier. Moreover, there is another discernible opportunity here, which is the pipelining of responses and instructions. Some aspects of this and other emerging characteristic of Kafka are explored below.

As when loading the sequences to Kafka, Kubernetes is used to deploy the GC Content agent with increasing levels of parallelization. In each case the number of
GC Content agent

This is a small autonomous program that listens for GC Content instructions and then performs a GC Content transformation on the RefSeq messages.

GC Content Instruction partitions was set to be the same as the level of parallelization.

The measurements taken for GC Content using Field of Genes were used to gauge if this architecture led to improved performance and scalability. The benchmark in the case of GC Content was an explicitly multi-threaded, single-node solution, using a single instance of precisely the same server configuration. For the same reasons given in the description of the Loader Agent benchmark, this is considered a valuable comparison. Again, the measurements are made by increasing the parallelization factor (in this case, the number of threads) and the amount of data to be processed, in tandem, looking for a flat system response.

The method of measurement for Field of Genes in this case is slightly different to the Loader measurement. Streams are effectively infinite sources of data. In order to know when a stream is complete, two options present themselves: either by placing back-markers in each partition to indicate that the end has been reached, or by waiting for the system to reach a kind of equilibrium where the output is no longer changing. This notion of equilibrium is discussed below.

The latter approach was chosen for our experimental measurements. Using Kafka’s administration API, the size of the output topic is regularly measured. The stream was considered complete when its size didn’t change in a defined period of time - in
our case 10 seconds.

5.4. Results

5.4.1. Downloads: Benchmark vs Field of Genes

Table 5.1 shows the downloading time in seconds for the Benchmark ($DL_b$) and Field of Genes with 8 and 12 servers in the cluster respectively ($DL_{FoG-8}$ and $DL_{FoG-12}$). Each row shows the results for the same fixed number of RefSeq files and level of parallelization - threads in the case of the Benchmark and agents in the case of Field of Genes ($T/A$). Note that the last two rows of the Benchmark are empty as the single server did not have enough space to store 20 files or more. In the case of the Field of Genes, what is being measured includes the loading of the sequences from the downloaded files into a Kafka topic.

When these download values are plotted with parallelization factors along the X-axis and download time on the Y-axis, as shown in figure 5.6, we can compare how well the two options scale up. A flat horizontal line represents perfect scalability where the overall time to download does not change when extra data is added, as long as the parallelization factor increases to the same degree. In this kind of plot, the scalability can be seen to be inversely proportional to the slope of the line. The

<table>
<thead>
<tr>
<th>$T/A$</th>
<th>$DL_b$</th>
<th>$DL_{FoG-8}$</th>
<th>$DL_{FoG-12}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>112</td>
<td>301</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>176</td>
<td>301</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>247</td>
<td>302</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>323</td>
<td>301</td>
<td>302</td>
</tr>
<tr>
<td>20</td>
<td>302</td>
<td>302</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>302</td>
<td>302</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.1.: Downloading times (s)

Benchmark and Field of Genes with 8 and 12 servers in the cluster respectively, against levels of parallelization ($T/A$).
5.4.2. GC Content: Benchmark vs Field of Genes

The same formats of data (table 5.2) and plot (figure 5.7) are presented for the case of the GC Content processing. In this case, we see a departure between the 8-node and 12-node cluster behaviour. The limits of scalability for the smaller cluster are reached between 16 and 20 RefSeq files, for reasons discussed later. The larger cluster however is able to extend scalability further.

Whereas the previous plots are designed to show scalability, figure 5.8 compares the raw performance of the Benchmark and Field of Genes systems using sequences per second as a metric (where sequences are strings of genetic code up to 100,000 characters long). In this format, the greater the value on the Y-axis, the more performant the system.
Table 5.2.: GC Content times (s)

Benchmark and Field of Genes with 8 and 12 servers in the cluster respectively, against levels of parallelization (T/A).

<table>
<thead>
<tr>
<th>T/A</th>
<th>(GC_b)</th>
<th>(GC_{FoG-8})</th>
<th>(GC_{FoG-12})</th>
<th>Sequences</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>283</td>
<td>368</td>
<td></td>
<td>156,086</td>
</tr>
<tr>
<td>8</td>
<td>552</td>
<td>383</td>
<td></td>
<td>309,703</td>
</tr>
<tr>
<td>12</td>
<td>810</td>
<td>380</td>
<td></td>
<td>458,288</td>
</tr>
<tr>
<td>16</td>
<td>1,101</td>
<td>381</td>
<td>368</td>
<td>616,215</td>
</tr>
<tr>
<td>20</td>
<td>506</td>
<td>379</td>
<td></td>
<td>769,525</td>
</tr>
<tr>
<td>24</td>
<td>562</td>
<td>432</td>
<td></td>
<td>926,824</td>
</tr>
</tbody>
</table>

Figure 5.7.: Scalability of GC Content

Flat lines indicate ability to scale, while higher slopes are inversely proportional to scalability. Increasing the number of nodes improves ability to scale.
Figure 5.8.: Processing Rates of GC Content (seq/sec)

In this plot, the y-axis is measuring throughput. Straight-line positive slopes indicate ability to scale. Flat lines indicate inability to scale.
5.5. Discussion

The results presented in the previous section lead to a number of conclusions.

5.5.1. Download

When downloading data from NCBI, the two alternatives show very different characteristics. The Benchmark solution is initially quicker to download than the Field of Genes, but that performance deteriorates rapidly as more data is added (despite additional threads being made available). The Field of Genes performance remains almost perfectly constant, and outperforms the Benchmark when 16 RefSeq files or more are downloaded.

The gap in performance at lower levels of data is due entirely to an extra step required of the Field of Genes: While the Benchmark process downloads, unzips and converts the RefSeq files from NCBI, the Field of Genes process does all this and then writes the sequences to Kafka.

If we subtract the time needed for this extra step, Field of Genes behaves more efficiently even for lower numbers of files. This is shown by the dotted line (marked Adjusted) shown in figure 5.6. When we remember that the data in Kafka is available as a stream, and therefore can be accessed and processed as soon as it arrives in Kafka, we can say that this dotted line is a more valid comparison with the Benchmark when considering the performance of multi-stage processing pipelines.

5.5.2. GC Content

The GC Content processing presents a similar picture, but two differences stand out.

Firstly, a much steeper slope (151.5 compared to 17.6) is in evidence in the case of the GC Content Benchmark when compared to the download Benchmark, indicating greater challenges in scaling the GC Content process than the download. It can be
reasonably guessed that this is due to the bottleneck of processor availability, which is tighter than the network bottleneck that prevailed during download.

Secondly, the Field of Genes scalability also reached a limit, albeit a softer one and a later one. The crucial thing to point out here is that the scalability bottleneck of Field of Genes was addressed by extending the Kafka cluster from 8 nodes to 12. This ability to arbitrarily extend size and scalability is one of the features that makes Kafka such a suitable repository for bioinformatic data.

5.5.3. Emerging Characteristics

Implementing any software system involves a certain amount of on-the-fly design. One can never know what the complete solution will be until the finer complexities have been encountered and dealt with in the code itself. This is what is meant by the term Emergent Design (Bain 2008), and it is a useful exercise to look back on any implementation, especially proofs of concept, in order to see what else can be learned from the experiment, beyond the original hypothesis.

In the case of Field of Genes, two unanticipated features are pointed out, which may be worth building on.

Firstly, note that a system that uses Kafka to store data will also tend - for expediency - to use Kafka to store instructions. This is especially the case when large numbers of autonomous agents operating on the data need to be coordinated. An emerging feature of this tendency is the ability to pipeline not only the data, but also these instructions, so that the results from one agent might trigger the behaviour of another. Rather than spending too much time predicting where this may lead, it is enough for now to point out something which every biologist knows: that from many small and simple co-operating elements, very complex and intelligent pathways may be constructed.

Secondly, another feature of stream-based programming, which has been touched on when describing measurement in section 5.3.4, is the idea that a process in some
sense may never be finished, but instead arrives at equilibrium, at which point the most recent results may be read off a final topic. As new source data are fed upstream into such a system, they create a ripple of computation resulting in a refreshing of the final results. While this is not the behaviour that is typically expected of software systems, in the era of big genomic data which is constantly changing, it may become accepted as a suitable paradigm. The approach has a precedent in software engineering where it is known as *eventual consistency* (Bailis & Ghodsi 2013).

### 5.5.4. Orchestration using Kubernetes

For many experiments, using a single Docker container can be sufficient. However, in complex arrangements of executables, such as the distributed, cloud-based Apache Kafka experiment described above, it is not always enough to be able to execute single components. The entire system of executables must be correctly orchestrated in order to reproduce the result. This orchestration includes the correct configuration and execution of third-party components as well as those developed as part of the experiment. A number of solutions have emerged to provide this kind of orchestration, particularly addressing the situation where the individual components are containerised. *Kubernetes* is foremost amongst them.

High-scale and/or high-complexity scenarios are regularly encountered in software engineering projects and Kubernetes\(^m\) has emerged as a standard way of orchestrating multiple containers into a single system (Burns et al. 2016).

Kubernetes evolved from Google’s in-house container orchestration system, known as Borg (Verma et al. 2015). It was first released in 2015 and since then has become the *de facto* standard in Docker container orchestration, offered as a product on multiple cloud vendors like Microsoft’s Azure, Amazon’s Amazon Web Services (AWS) and Google’s own GCP.

\(^m\)https://kubernetes.io
Kubernetes, often shortened to **k8s**, can be understood in a number of different ways. Two useful interpretations are:

**Abstraction over computational infrastructure:** Complex distributed systems are implemented in terms of elements such as VMs or nodes, load balancers, reverse proxies, SSL terminators, virtual networks, services, batch jobs, role-based access systems and more. While every system (either cloud-based or on-premises) will be composed of one or more of these elements, the details of their implementation will vary greatly. Kubernetes provides abstractions of these elements, and hides that implementation. A system designed to run on Kubernetes, will run on an on-premises canonical Kubernetes installation as well as on a cloud-based implementation like Azure Kubernetes Service (AKS).

**Declarative system orchestration:** Kubernetes represents an example of Infrastructure as Code. Rather than deployment via imperative-style scripting, Kubernetes starts with a description of the *desired state* - a set of YAML-file declarations of the kinds of system components outlined in the previous paragraph. A Kubernetes component called the *kube-controller-manager*\(^a\) constantly compares this desired state with the *actual state* and adjusts the system to bring them into alignment.

YAML files represent the desired state of the running system, in terms of infrastructure abstractions. These files can then be stored in source control like any other code file, and like this, the architectural and process aspects to reproducibility unite: Reproducible source code gives rise to reproducible deployment artifacts, and reproducible infrastructure code gives rise to reproducible deployments.

A public GitHub project\(^b\) for the Apache Kafka scalability experiment mentioned previously, contains all of these elements:

1. Source code for each of the deployable components.

---

\(^a\)https://kubernetes.io/docs/reference/command-line-tools-reference/kube-controller-manager/

\(^b\)https://github.com/blawlor/field-of-genes
2. Build instructions to create a deployable artifact for each (a Docker image in all cases).

3. A set of Kubernetes YAML files describing what the Apache Kafka cluster and experimental subsystems should look like once deployed.

4. Instructions for rolling this out on public cloud infrastructure.

The experiment is a complex, cloud-based, distributed system using a variety of languages and technologies. It was first run in October of 2016, and remains reproducible - in principle on any major public or private instance of a Kubernetes cluster - at the time of writing.

5.5.5. Future Work

Having interpreted the data, and described some of the emerging characteristics of Field of Genes, it is appropriate to ask how this approach might be used by the bioinformatic community.

From the perspective of the software engineer, Field of Genes presents a fast and scalable access point to raw data whose structure and format is independent of any particular algorithm (and therefore open to all). It is an open-ended system which can accept updates in real time and propagate those updates to any consumers. Given all this, I have presented Kafka as a suitable primary central repository from which bioinformatic data could be published. Algorithm-specific databases such as BLAST, structured for a specific purpose, could then be constructed downstream of Kafka.

The specific way in which bioinformatic data should be stored as messages on Kafka topics requires further study, and should take into account the specific nature of that data. For example, entire genomes would need to be split across many messages, in a way that permitted parallel access while preserving the order. Short
Figure 5.9.: Suggested use context of Kafka and Akka.

Kafka as a primary repository of bioinformatic data, and also containing secondary data transformed by Akka actors. Other more highly-structured data constructed from primary data.

reads - the output of Next Generation Sequencing processes could be stored one-to-the-message with no concern for order.

Besides being a suitable place to store data, Field of Genes has demonstrated itself to be a suitable platform on which to process those data, most particularly in the case where all data are to be processed. The clustering infrastructure that hosts Kafka can also serve as a computational fabric on which to process that data. Alternatively, a separate cluster comprised of computation-only nodes could be created. The technology used for the agents - Reactive Streams using Scala and Akka - as described in chapter 4 - were designed specifically with Kafka in mind.

Figure 5.9 represents this general approach.
6. Case Study: Democratising Bioinformatics

A version of this chapter was published as The Democratisation of Bioinformatics: A Software Engineering Perspective (Lawlor & Sleator 2020) in GigaScience (Oxford Academic) in 2020 where it has so far accumulated an Altmetric score of 26, and more than 1000 views.

The paper, which argues for an explicit role for software engineering in bioinformatics, was the product of a period of 5 months in 2016 spent working in a molecular diagnostics company as Software Development Manager together with subsequent related work in the years that followed. The core staff at the time was composed of bioinformaticians, sales (with biology background), C-level managers with scientific and computing backgrounds, and some software developers, mostly outsourced.

During this period, I put a set of software engineering tools and processes in place to facilitate collaboration between software engineers and bioinformaticians with the aim of creating scalable and reproducible bioinformatic pipelines.

• **Views:** 1458

• **Altmetric:** 36

• **Reference:** GigaScience 9(6), doi 10.1093/gigascience/giaa063

---

aNSilico Lifescience Ltd.
The output of this research also included new bioinformatic pipelines developed in collaboration with specialised bioinformaticians using Docker as the crossover technology. In some cases the pipelines were integrated into NSilico’s Simplicity platform (Walsh et al. 2013), enabling scalable access to these pipelines for the wider bioinformatic community.

The 10 bioinformatic publications that were produced as part of this research are presented in section A.2. Collectively, they have so far garnered 20 citations and more than 2500 reads.

6.1. Background

There has perhaps never been a more potentially productive time to be a software engineer. Thanks to a number of factors, which collectively can be referred to as democratisation of software at scale, it is possible for relatively small teams of engineers to produce internet-scale products (i.e. software systems that scale globally), a feat which was previously the exclusive preserve of large organisations. The key to this progress has been abstraction.

The idea that abstraction should be key to simplifying the development of software will come as no surprise to computer scientists. To quote Liskov & Guttag (1986), “[a]nyone who has introduced a subroutine to provide a function that can be used in other programs has used procedural abstraction”. They go on to explain how abstraction hides “irrelevant’ details, describing only those details that are relevant to the problem at hand”. Devlin (2003) frames the importance of abstraction by saying that “computing is all about constructing, manipulating, and reasoning about abstractions.”

Students of Computer Science will recognise how abstractions can be layered, one on top of the other, as in the Open Systems Interconnection (OSI) standard (Zim-
In a comparable way, entire fields or competencies have been layered on top of each other in modern software engineering. The democratisation factors mentioned above include containerization, orchestration and cloud computing, and they share a common theme: abstracting away the accidental complexity of a problem and leaving only its essential complexity exposed (Armbrust et al. 2010). In particular, they hide much of the complexity of network engineering, cluster management, and running distributed systems reliably and at scale. This empowers software developers to concentrate on their core domain: providing features to users.

However this stratified abstraction approach has not yet been widely applied in bioinformatics software development. During the research period outlined in the preamble to this chapter, which entailed close collaboration between software engineers and bioinformaticians, it was observed that the day-to-day experience of many bioinformatic researchers and practitioners was often one of frustrations, delays and impediments to productivity. It became clear that these negative experiences were due in part to having to work at the wrong level of abstraction, dealing with implementation details that merely distracted from the work at hand, and being obliged to improvise software solutions that subsequently presented problems in terms of scalability and reproducibility. These observations are reflected in the literature (Grüning et al. 2018).

Accordingly, much of the effort expended in the related period of research was to identify ways in which the advances in software engineering, outlined in previous chapters, might be made more readily available to bioinformaticians. The premise was that, in the same way that access to distributed systems engineering has been democratised for generalist software engineers, access to scalable and reproducible software engineering could be democratised for generalist bioinformaticians and biologists.
In a highly regarded software engineering paper *No Silver Bullet* (Brooks 1987), Fred Brooks wrote of the difference between “essential tasks” and “accidental tasks” in software. Essential tasks, according to Brooks, relate to the fashioning of conceptual structures that make up the abstract software: analysing and modelling the problem domain. Accidental tasks, by contrast, are about implementing these abstractions in real programming languages, on real computers, with real resource constraints. While Brooks’ observations are old, they are certainly not dated. As he predicted, no “silver bullet” has presented itself in the intervening decades to significantly reduce the *essential* complexity of software development. Brooks’ observation that most of the progress made in software productivity have come from “removing artificial barriers that have made the accidental tasks inordinately hard” remains true.

But what is “accidental” to one discipline is “essential” to another. The complexities of creating a distributed computing environment - networking, security, reliability, elasticity - are “accidental” for generalist software developers, but “essential” for cloud providers like AWS, Azure and GCP who simplify such environments for those developers. Cloud computing has evolved over the years from providing Infrastructure as as Service (IaaS) to offering *Platform as a Service* (PaaS). Rather than merely selling time on Virtual Machines, cloud providers have opted to provide entire platforming solutions such as relational databases, lambda function support and Kubernetes clusters (Mell & Grance 2011). This has freed generalist software developers to concentrate on their *essential* complexity; the modelling of solutions using scalable architectures.

In bioinformatics systems, the accidental tasks are those which require these very software engineering skills and techniques, which are additional to the ever-increasing complexity that already exists in the biological domain. The burden of these accidental tasks, in the face of greater demands for scale, and mounting concerns around reproducibility, is widely felt. These are the “artificial barriers” of
Figure 6.1.: Roles and their interfaces in bioinformatic software development.

Roles are defined by their level of abstraction. Each role provides a service to the one above it. Handover artifacts are identified at the boundary between each layer.

software engineering which are “inordinately hard” for generalist bioinformaticians.

In figure 6.1 we take a step back and look at the enterprise of creating modern, scalable, cloud-native bioinformatic applications in a wider context. A useful way to view the relevant roles and their relationships is presented, which emphasises the layering of abstractions in which the accidental in one domain, is essential in another. It identifies ways of interacting at the boundaries of these roles, which we will discuss next.

The figure coins the acronym EaaS to indicate the Engineering as a Service that generalist software engineers, standing on the shoulders of PaaS, can in turn offer to bioinformaticians. Similarly, bioinformaticians can blend their understanding of computation and biology into applications and pipelines (Software as a Service - SaaS) that can be easily used not only by other bioinformaticians but by all
biologists. The work of clinicians and researchers can be seen as Biology as a Service - to academia and to society.

This, then, is an outline of how the layering of abstractions simplifies and thereby democratises access to internet-scale computing for software generalists, for bioinformaticians and for life scientists.

The breadth of engineering knowledge required to do reproducible bioinformatic work at scale is perhaps not fully appreciated (Storer 2017). Such skills cannot be absorbed in their entirety by bioinformaticians and other scientific programmers. In order to create bioinformatic systems of scale, there are different kinds of complexity that come into play, which fall well outside what should be considered as the essential tasks of the bioinformatician. The work done and presented on such topics as concurrent programming techniques and reproducible build and deployment methods in previous chapters and sections, are good examples of this complexity.

Current bioinformatic practices and attitudes are likely influenced by the latent assumption that, because bioinformatics is already a mix of biology and computation, there is little or no call for software specialists. There is also the view held by some, but without much evidence, that software engineers cannot work alongside scientists for various reasons including complexity, process and budgets. These themes were treated in subsection 2.2.2 of the literature review.

As this chapter will argue, such collaboration is not only possible, it is necessary. The key is knowing where to draw the boundary between the disciplines, and what information or artifacts should cross that boundary.

6.2. Methods

This part of the research is based on 5 months in 2016 spent working in a molecular diagnostics company called NSilico Lifescience Ltd. as Software Development Manager. The principal activity during this time was the further development of
NSilico’s flagship software platform called Simplicity\(^b\) - by abstracting away many technical aspects of bioinformatic pipeline creation, while managing all aspects of scalability and reproducibility. Simplicity was introduced in Walsh et al. (2013) and it is described in more detail below.

As part of the development of bioinformatic pipelines for Simplicity, Docker technology was introduced to address a number of needs - in particular, those related to reproducibility and scalability. What emerged as part of this work was that Docker also addressed the question of where to draw the boundary between software engineering and bioinformatics, and what that boundary might consist of.

Docker is now widely used in the life sciences (Menegidio et al. 2017, da Veiga Leperivost et al. 2017), though when this work was being done, using Docker was still novel (Walsh et al. 2016b, 2017b), if one considers that the first papers to describe its use appeared in 2015 (Moreews et al. 2015, Aranguren & Wilkinson 2015). The technology itself has been described in chapter 2 and it has been presented elsewhere in this thesis as it pertains to reproducibility and scalability. It is presented here again, but this time as a candidate crossover technology between software engineers and bioinformaticians. By specifying in code form (the Dockerfile) exactly what a container should contain, then questions of Linux distributions and versions, system configurations, installed libraries and tools, directory structures, environment variables and many other elements can be specified, built and tested by a software engineer. This specification can be use to create running Docker containers on which bioinformatic pipelines can be developed and tested by a bioinformatician, using the tools installed. This makes use of Docker’s lightweight virtualization nature.

The Docker container can also represent an agreed series of inputs and outputs that an engineer can integrate into a scaled and distributed implementation of the bioinformatician’s encapsulated process.

This precise approach was used to good effect when developing many of the

\(^b\)See https://simplicity.nsilico.com/
pipelines cited in section A.2. The remainder of this chapter describes that development experience, with a view to underlining the feasibility of engineer-bioinformatician collaboration, with Docker at the boundary. Where chapters 4 and 5 have given examples of software process and architecture solutions to scaling computation and data, this chapter examines processes and architectures that enhance reproducibility and - vitally - demonstrate how to integrate the expertise of the software engineer with that of the bioinformatician. This is done in the context of the Simplicity platform under development in NSilico at the time.

6.2.1. Simplicity: Bioinformatic Pipelines

Bioinformatic pipelines are a mainstay of modern biological research and practice. Typically a pipeline consists of a series of Linux-based commandline tools, connected to each other so that the output of one acts as the input to the next. Some examples of such pipelines will be described below. Creating them can be “labour intensive, error prone, untraceable and often result in the generation of significant amounts of data” (Walsh et al. 2013).

The goal of the Simplicity platform is to make it easier for biologists and bioinformaticians to access and run such pipelines. It achieves this by abstracting away much of the accidental complexity involved in their creation and also by taking responsibility for their execution on suitable hardware.

The intended result is a democratisation - an opening up of access - with regards to these pipelines, allowing researchers and practitioners with limited technical know-how and/or access to scalable hardware, to analyse their data quickly and reproducibly.

It was recognised within NSilico that adding new bioinformatic pipelines to Simplicity was often difficult, as it required a combination of the disparate skills of bioinformaticians and software engineers, the former being typically specialists in the type of pipeline being added, and the latter being specialists in the design and
implementation of the Simplicity platform. Additionally, the Simplicity backend was deployed onto on-premises, high-performance hardware rather than cloud-based, commodity servers. This was considered to be a limitation to scale.

In order to enhance Simplicity’s ability to add new pipelines, and also to make it more scalable through deployment to the cloud on commodity hardware, some key changes to tools and processes were introduced.

1. Rolling out software engineering tools and practices which are considered standard in the industry: Best practices of source control were put in place (feature branches, issue-management, pull requests, continuous integration)

2. All new pipelines would be developed and deployed within Docker containers, following the reasoning outlined in the previous section, summarised by figure 2.7.

The following section describes these pipelines, their implementation details, and the tools and processes involved.

6.2.2. Software Processes in the Service of Bioinformatics

Various SCM tools and processes have been created through the discipline of software engineering in order to meet the requirements of reproducible code, builds and executions, and these have been presented in the literature review in section 2.4.1. These source control systems and standardised build tools including dependency management, are considered essential to the task of software engineering.

However, according to the survey by Lawlor & Walsh (2015) presented in chapter 3 and further reflected by other surveys (Hannay et al. 2009, Nguyen-Hoan et al. 2010), scientific software developers underestimate the importance of SCM tools like version control, or don’t make use of them at all.

The survey in chapter 3 identifies a gap: while more than 90% of software engineers strongly agree with the statement that “source control is used in your organisation’s
projects”, only around 50% of life scientists agreed.

Moreover, although 92% of software engineers indicate that build systems are used in their organisation’s projects, life scientists are much less familiar, with less than 40% indicating build tool use. For more details see figure 3.1.

The problem is widely recognised in the literature, with papers like Peng (2011), Ioannidis et al. (2009) and Sandve et al. (2013) describing scenarios where code or data is no longer available, or technologies are not understood well enough to reproduce experiments.

The role of software engineers in the creation of SCM processes and tools, constitutes the Engineering as a Service (EaaS) presented in figure 2.7. In the course of this research period, the engineers chose, maintained and supported the source control and build infrastructure on behalf of the bioinformaticians.

6.2.3. Software Process Abstractions

What follows is a description of the software engineering concepts and their implementations that were used during the collaboration in NSilico, and an analysis of their use in a mixed team of software engineers and bioinformaticians.

Scrum and Issue Management Abstractions

Scrum is a commonly-used process in software engineering. As a prelude to explaining its function and purpose, we need to define some terms:

Product Owner A role within the organisation, responsible for giving a given product a strategic direction. Product owners decide what functionality is needed and sets priorities. They do this by means of filling and maintaining the product backlog.

Issue There are three kinds of work identified in Backlogs - Stories, Tasks and Bugs. They differ slightly in their scopes, but have in common that they must be
doable within a single *sprint* and have a clear *definition of done*.

**Product Backlog** A prioritised list of *issues* to be scheduled for execution by the development team.

**Sprint** A timebox (typically two or three weeks) in which a fixed number of work items have been scheduled to be completed.

**Definition of Done** An unambiguous statement of the content of a given *issue*, expressed in terms of the expected outcome.

Although defined in terms of a *product*, the vocabulary above can be applied to any shared goal or project. Any such endeavour needs strategic direction and operational planning. Where the Scrum literature talks about *development teams*, we can interpret this more loosely as any team of collaborating engineers and scientists.

Issue management is a project management discipline and therefore in our case related to the use of Scrum. But even where Scrum is not used, software engineering efforts are typically managed by breaking down the overall effort into smaller tasks and guiding these tasks through to completion as *issues*.

An issue contains the following information related to the work it describes:

1. An unambiguous identifier for the work (numeric, textual or both)

2. A full description of the required work, completed by a Subject Matter Expert (SME) and written in a form that can be used by the implementer

3. An estimate of the amount of effort required to complete the issue

4. The name of the implementer to whom the issue is currently assigned

5. A “conversation” that tracks questions and answers between the implementer and the SME

6. A log of time spent on the issue
7. The current state of the issue. The permissible states of the issue can vary from context to context, but in general they cover a lifecycle of states, typically starting with the To Do state and ending with Done.

By creating, updating and tracking issues, teams can more easily communicate their intentions and progress.

Source Control Abstractions

The correct use of source control is fundamental to the day-to-day work of a software engineer. It can be considered as a type of hygiene practice necessary for the continuing health of the shared code. Here again we define some important terms for later use.

Source Control A Source Control system enables developers to keep historical versions of source code and project files that are under development and thereafter to retrieve past versions (Ruparelia 2010).

Distributed Source Control (DSC) A source control system in which every user has an entire copy of the repository on which they are working. The most popular example of this at the time of writing is Git (Spinellis 2012).

Repository The smallest versionable/branchable unit of code in a DSC. Typically also the unit of release.

Branching A feature of most modern source control systems that allows multiple work items to be carried out independently and in parallel on the same repository without interfering with each other. Branches are usually merged back into a common root or mainline branch.

Branching Strategy There are many ways to use the branching feature (when and why to branch, when and how to merge). GitFlow (Dwaraki et al. 2015) and derivatives like Branch-per-Feature are common examples in Git and other
DSCs, which advocate creating branches for every identified issue (in the sense described in the Scrum description previously). These strategies generally include a naming convention for the branches created.

**Pull Request** As mentioned above, branches are usually merged back into a mainline branch and this can either be done directly by the developer who is working on the branch, or by a developer responsible for the quality of merges to the mainline branch. This latter approach involves the creation of a Pull Request - a request for merge. This allows for the code to be reviewed before being allowed into the mainline branch.

### 6.2.4. Software Process implementations

Activities were organised into sprints as described in the previous section. Initially we chose sprints of two weeks’ duration with a view to releasing new functionality at the end of each sprint. That new functionality typically took the form of a new bioinformatic pipeline, either partial or complete. During the project, 3 new pipelines were completed:

- TB Resistance Profile Pipeline
- Metagenomics Pipeline
- C.diff Resistance Profile Pipeline

Pipelines were composed of various Services - custom code designed to interface with the overall architecture of this project. Services called lower-level Tools - usually opensource bioinformatic tools like samtools\(^c\) or qiime\(^d\).

\(^c\)http://samtools.sourceforge.net/
\(^d\)http://qiime.org/
Scrum and Issue management Implementation

From an operational point of view, a cloud-based tool called Jira was used to implement the Scrum process described above. Product Owners used Jira to create new projects and to maintain the backlogs of existing projects. Developers and bioinformaticians used it to plan and execute sprints, and to track progress on individual issues.

Jira has many integration points with the chosen Git source control system:

- By creating a feature branch with the ID of the issue, Jira is able to maintain a list of all branches associated with that issue (where the issue requires changes to multiple repositories) and automatically changes the state of the issue to “In Progress”.

- By using the ID of the issue in the message of every Git commit, Jira is able to maintain a list of all code changes associated with that issue.

- When a PR is created for a given branch, the associated issue changes state to “In Review”.

- When a PR is accepted and merged, the associated issue changes state to “Done”.

This tool and its integration points support the Branch-per-Issue strategy very well.

Jira issues become a central resource. They are used to maintain notes on the work being carried out, not unlike a laboratory notebook. As explained above, the integration points between Git and Jira mean that all code changes, branches, Pull Requests and merges associated with the issue are automatically linked to that issue in Jira. Time spent on an issue can also be logged using Jira, which can help track overall progress and plan remedial action where necessary. The overall effect of using

---

*https://www.atlassian.com/software/jira*
a system like Jira is to create an information radiator\textsuperscript{7}, providing a greater degree of transparency and predictability across the organisation.

**Git Implementation**

The Branch-per-Issue source control branching strategy was selected. That is to say, for every identified work item during a sprint that required changes to code or documentation, a branch was created in Git\textsuperscript{9}. This allowed each team member to work in isolation from others, while still committing and pushing their work back to the central source code repository. When work on a given item was complete, the team member would create a Pull Request (PR). The code in each PR was reviewed by a Lead Engineer and either approved and merged to the develop branch, or annotated for further change (to be done by the original developer).

**Containerization Implementation**

We used Docker in two main ways. Firstly, during development, we used Dockerfiles and their associated images as vehicles for software engineers to share partially complete services with bioinformaticians in order to check for biological correctness and to run tests. Services were heavily dependant on the correct installation of a variety of tools and utilities, but the use of Docker images removed the need to recreate these dependencies on the part of the bioinformatician. This was an important factor in removing bureaucratic toil from the workload of the bioinformaticians, allowing them to focus on the biology.

The simple text Dockerfile is easily shared and updated over time. When the bioinformatician hits a technical problem, the software engineer can reproduce it, investigate it and fix it, and then send an amended Dockerfile back to the bioinformatician. When the bioinformatician has finished, the Dockerfile becomes the means by which a Docker image is created, distributed and run by any other users.

\textsuperscript{7}http://alistair.cockburn.us/Information+radiator

\textsuperscript{9}A cloud implementation of Git called Bitbucket was used.
This leads into the second way Docker was used. When releasing new functionality, it was possible to deploy the completed Docker images into production. This not only reduced the time and effort spent on deployment, it also ensured a continuity - what has been tested in development was the same as what was being deployed into production.

As well as this double element of reproducibility, Docker also added an element of traceability to the bioinformatics pipelines being developed. The Dockerfiles ensured that the execution environment - what tools and utilities were installed and which versions - was explicitly documented as part of the code.

Other scholars concur that containerization is a valuable strategy for enhancing reproducibility in bioinformatics. Since the Docker research presented in this thesis was first published in Walsh et al. (2016b) and again in Mac Aogáin et al. (2017), researchers such as Menegidio et al. (2017) have stated that the “[e]mergence of the Docker project is providing a promising new strategy to tackle these problems by enabling the configuration of a complete computing environment, in which all the information required for a particular operation can be implemented in a single container, which can be consistently exchanged and deployed in different platforms, regardless of the specificities of their hardware and/or operating systems (OS)”.

Beaulieu-Jones & Greene (2017) combined Docker with the software engineering technique of Continuous Integration to reproduce computational research.

It can be noted from this, not only that Docker containerization provides a good architectural basis for reproducible software in bioinformatics, but that it can be used to good effect in a bioinformatic Software Development Life Cycle (SDLC).

In both cases, not only was the build artifact (a Docker image) intrinsically easier to run on other environments but the instructions on how to build it were stored in source control as part of the program code. Thus, by adhering to best practices with regard to process (tracking all code in source control) bioinformatic projects that leverage containerization can enhance reproducibility of build and execution.
In the Apache Kafka experiment described in chapter 5, the build process includes a final step in which a Docker image is created from the executable code. The Dockerfile from which this image is generated prescribes the runtime environment expected by the executable. This Dockerfile can be seen both as documentation for that runtime system, but also as a reproducible recipe to create it. As a result, each element of that experiment enjoys reproducibility of execution.

6.3. Results

6.3.1. Collaborative Models

In Lawlor & Walsh (2015), the case was made that “the most effective way of introducing software engineering values into bioinformatic research is to introduce software engineers themselves”. In response, Dahlquist et al. (2016) disagreed, showing that “best practices can be taught to undergraduates concomitant with training in bioinformatics”. Taschuk & Wilson (2017), while recognising the problems presented in Lawlor & Walsh (2015), countered that “you don’t need to be a professionally trained programmer to write robust software”.

There is nothing incorrect in either of those responses but they are incomplete. As proposed in the Lawlor and Walsh paper - and repeated in this thesis - bioinformaticians should be sensitised to the importance of software engineering skills. But this is not sufficient. Dahlquist et al. and Taschuk and Wilson do not take into account the breadth of engineering knowledge required to do reproducible bioinformatic work at scale. Such skills cannot be absorbed in their entirety by bioinformaticians and other scientific programmers. In order to create bioinformatic systems of scale, there are different kinds of complexity that come into play.

To use the terminology from Brooks (1987), in bioinformatics systems, the accidental tasks are separate from and additional to whatever essential complexity exists in the biological problem domain. The burden of these accidental tasks even-
tually demands more of the scientific programmer than their necessarily peripheral training in software engineering can provide.

Other authors have expressed concerns about integrating software engineers into scientific teams and workflows. Segal & Morris (2008) characterise the software development process as starting with “[f]ull up-front requirement specifications”.

However this is based on outdated ideas of how software engineering works in commercial settings. Modern agile software engineering processes are explicitly designed to take account of the fact that requirements cannot be known up front and must be arrived at by iterative steps and in collaboration with a SME. Moreover, it is the core competence of a software engineer to understand *enough of* the problem domain to model the salient aspects in code. This is one of the reasons that software engineers tend to specialise in industry verticals (which change slowly) rather than in technology stacks (which change quickly). Kane et al. (2006) have reported that “agile methods are well suited to the exploratory and iterative nature of scientific inquiry.”

Where long term studies have been carried out on collaboration between biologists, bioinformaticians and software engineers, as in the case of Letondal & Mackay (2004), more nuanced conclusions are drawn: “[C]rossing these disciplinary boundaries is difficult. Individual biologists and programmers must position themselves along a continuum with respect to their technological skills... It is tempting to require that everyone become a bioinformatician, with extensive training in both domains. However, this is very expensive and does not take optimum advantage of individuals’ capabilities and interests. Also, even bioinformaticians constantly struggle to stay current with the technical advances in both disciplines”.

When it comes to scientific problems, Segal & Morris (2008) suggest that “[t]he average developer just doesn’t understand the application domain” in the same way that they might understand the needs of an “accounting package”.

This opinion suggests a lack of understanding of the role of the software engineer.
It is not necessary for the engineer to acquire domain expertise to the same level as the scientist. Software engineers are skilled in acquiring *enough* knowledge of a given domain to model that domain in software at an adequate level of abstraction for the problem in hand. As students, they are not taught about banking, insurance, complex financial instruments, aviation or medicine, but they can be found today working effectively in all these fields and more.

That said, every field is different and has its own particular engineering needs and problems. The kinds of process and technologies that work in an ecommerce project will differ from those of a telecommunications project. The same can be said of bioinformatic software projects. In any software project, as the process moves from the gathering of requirements, through a design stage, and into development, there is a handover point between the SME and the software engineer. Even in *agile* projects this handover is observed, albeit in regular cycles as part of an iterative process.

In most development settings this handover takes the form of requirements documents of varying degrees of detail.

Bioinformatics, however, is a field in which the SME is typically a scientist with coding skills, and as such, the handover point between SME and software engineer tends to be further along the development pipeline, and can even be in the form of code. This requires some adjustments to typical processes, and an important ingredient to these adjustments is the Docker containerization technology described in previous chapters.

### 6.3.2. Bioinformatic Output

Table 6.1 gives an overview of the 10 publications which can be considered part of the output of this research period, annotated according to whether the abstractions outlined above (Scrum, SCM, Docker) were applied.
<table>
<thead>
<tr>
<th>#</th>
<th>Publication</th>
<th>Authors</th>
<th>Scrum</th>
<th>SCM</th>
<th>Docker</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Visualizing Next-Generation Sequencing Cancer Data Sets with Cloud Computing</td>
<td>Walsh et al. (2016b)</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>2</td>
<td>Towards a host-pathogen integrated molecular diagnostic for bacterial infection in newborn babies</td>
<td>Forster et al. (2016)</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>3</td>
<td>MetaPlat: A Cloud based Platform for Analysis and Visualisation of Metagenomics Data</td>
<td>Konstantinidou et al. (2016)</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>4</td>
<td>FindR-TB: A cloud-based tool for antibiotic resistance prediction in <em>Mycobacterium tuberculosis</em></td>
<td>Walsh et al. (2016a)</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>5</td>
<td>Fourteen Draft Genome Sequences for the First Reported Cases of Azithromycin-Resistant <em>Neisseria gonorrhoeae</em> in Ireland</td>
<td>Mac Aogáin et al. (2017)</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>6</td>
<td>A Metagenomics Analysis of Rumen Microbiome</td>
<td>Walsh et al. (2017b)</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>Title</td>
<td>Authors</td>
<td>Role</td>
<td>Use</td>
<td>Support</td>
</tr>
<tr>
<td>---</td>
<td>-----------------------------------------------------------------------</td>
<td>------------------</td>
<td>------</td>
<td>-----</td>
<td>---------</td>
</tr>
<tr>
<td>7</td>
<td>Investigating antibiotic resistance mechanisms in <em>Clostridium difficile</em> through genome-wide analysis of phenotyped clinical isolates</td>
<td>Walsh et al. (2017a)</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>8</td>
<td>ImmunoAdept - bringing blood microbiome profiling to the clinical practice</td>
<td>Walsh et al. (2018)</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>9</td>
<td>Antimicrobial resistance and molecular epidemiology using whole-genome sequencing of <em>Neisseria gonorrhoeae</em> in Ireland 2014 - 2016: focus on extended-spectrum cephalosporins and azithromycin</td>
<td>Ryan et al. (2018)</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>10</td>
<td><em>Simplicity DiffExpress</em>: A Bespoke Cloud-Based Interface for RNA-seq Differential Expression Modeling and Analysis</td>
<td>Palu et al. (2019)</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 6.1.: Pipelines Publications and their development characteristics

Each paper, my role in it, and its related pipeline where applicable, is outlined in the following section.
1. Walsh et al. (2016b) presents the use of the Simplicity platform in the visualisation of sequencing data sets related to cancer research. The paper was part of Sage Care\(^h\), a Marie Curie Horizon 2020 funded project. Of interest to this thesis is a description of how Docker is introduced to the back-end architecture in order to enhance scalability and reproducibility.

2. The hypothesis tested in Forster et al. (2016) concerns the ability to test for the presence of infection in neonates, based on the host immune response as measured by gene expression. My role in this paper, and its underlying Marie Curie project ClouDx-i\(^i\), was to maintain the bioinformatic pipelines required to assemble the sequencing output of the related pathogens from the infected neonates. These pipelines were made available as part of the Simplicity platform, which I was tasked to maintain and evolve with respect to reproducibility and scalability, as outlined in section 6.2.1.

3. Konstantinidou et al. (2016) relates the use of Simplicity as the basis for a Metagenomic analysis platform accessible to non-expert users, as part of the H2020 MetaPlat project\(^j\). The particular application described was the analysis and visualisation of rumen microbiomes, creating profiles to correlate with environmental factors. The principal tool used here was Quantitative Insights Into Microbial Ecology (QIIME) (Caporaso et al. 2010) which, although powerful and effective, can be difficult to install and use. Docker was key in “taming” QIIME, and this process was undertaken by me as software development manager and delivered in the form of a Docker container to the bioinformatician. Resolving difficulties in installing and running QIIME required an understanding of Linux package dependencies and also required a deeper reading of the tool’s documents and the forums that exist to support

\(^h\)www.sage-care.eu
\(^i\)http://www.cloudxi.eu/
\(^j\)http://www.metaplat.eu/
it. By transferring this requirement from the bioinformatician to the software
engineer, more time and energy was available to spend on the core biological
nature of the research.

Moreover, when the pipeline was complete, it was more easily integrated into
Simplicity, which by then had been converted to deploy pipelines as Docker
containers. This in turn made the pipeline more scalable - addition demand
for invocations of the pipeline could be addressed simply by instantiating more
Docker containers.

See also Walsh et al. (2017b) below.

4. Diagnosis and treatment of *Mycobacterium tuberculosis* (*TB*) in a clinical set-
ting presents difficulties and delays due to long lead times in cultivating sam-
pies and similarly slow direct tests for antibiotic resistance on those samples.
The research behind Walsh et al. (2016a) is concerned with creating a resis-
tance prediction algorithm based on a Whole Genome Sequencing of clinical
samples. It is anticipated that access to Whole Genome Sequencing data in
clinical settings would make such predictors viable as a diagnostic tool, rad-
ically reducing lead times. The specific details of this pipeline and a related
*Clostridium difficile* (*C. difficile*) version are presented below.

My role in this publication was the iterative development of the pipeline in
question, based on guidance from my bioinformatician colleague who acted as
an SME. During each iteration, I delivered the latest version of a Docker image
which the bioinformatician tested for correctness against biological criteria (in
particular, the sensitivity and specificity of the resistance predictions compared
to known results). The Docker images provided a stable and reproducible
platform of exchange, which together with a systematic use of source control
allowed both parties to collaborate safely and efficiently. The bioinformatician
was free to experiment with different approaches without having to deal with
the technical cost. He updated the database file, and I updated the code that accessed and interpreted that data, based on his instructions.

As with the previous example, the resulting Docker image was then available to integrate in a scalable manner into Simplicity.

5. Mac Aogain et al. (2017), an announcement of 14 draft genomes of Azithromycin-resistant *Neisseria gonorrhoeae*, used Simplicity to perform the sequencing assembly, analysis and reporting, facilitating and accelerating the announcement by simplifying the use of the sequence assembly pipeline. My role here was in the continued maintenance and development of the Simplicity platform.

6. Walsh et al. (2017b) builds on Konstantinidou et al. (2016) and in particular tests the features of dietary supplements and cattle breed as possible correlates with microbiome profiles. The *QIIME* Docker services created as part of Konstantinidou et al. (2016) were used to good effect. This work was also part of the MetaPlat project mentioned in that paper.

7. Walsh et al. (2017a) continues the work done in Walsh et al. (2016a) by extending the concept of genomic resistance prediction to *C. difficile*. Because this bacterium’s resistome consists not only of its inherited genome but also of horizontally transferred (non-chromosomal or plasmid) genes, the original pipeline for TB was extended to take this into account. A more detailed description of these pipelines is given later in this chapter.

8. Walsh et al. (2018) brings the question of microbiome profiling into a clinical context. A new pipeline was developed to provide an assay development platform for immunology. Simplicity’s role here once more is to provide an accessible pipeline to generalist users - in this case clinicians. The role of Docker was to simplify access to the bioinformatic tools described in the paper, and to enhance reproducibility. Once again, my role in this work was the creation and maintenance of the Docker container for the pipeline.
9. The collaboration on Ryan et al. (2018) was not related directly to Simplicity, but rather was based on Docker experience built up as part of working on Simplicity, and applied once again to the question of bacterial resistance to antibiotics. The pipeline in this case was a combination of Snippy$^k$ and Gubbins (Croucher et al. 2015), designed to run in a repeatable and efficient fashion over a number of fasta file pairs in order to build up a phylogenetic tree. Resistance was analysed with respect to membership of phylogenetic susceptibility categories. The goal of Docker in this context was to enhance the reproducibility of the experimental results.

10. Palu et al. (2019) represents a return to adding novel pipelines to the Simplicity platform, this time a differential expression pipeline. This was the result of a collaboration with the cited NSilico bioinformatician, and involved the use of R code$^l$ developed principally by the bioinformatician and integrated back into Simplicity using Docker. My contribution was of the same nature as that outlined in Walsh et al. (2016a) and Walsh et al. (2017a).

From the complete list of bioinformatic output above, we distil a smaller number of publications whose characteristics most strongly exemplify the main theme of this chapter: the utility of an explicit role for software engineering in bioinformatics, and the interaction between software engineers and bioinformaticians.

Research, like modern software development, is an iterative process (Dybå & Dingsøyr 2008, Kane et al. 2006). The desired goal is not always known in precise terms at the outset, and the exploration of both the problem and its potential solutions takes place in small and tentative steps.

The collaboration between software engineer and scientist must mirror this process, by simplifying the information exchanges that underpin the iterative process. Furthermore, the iterative approach of Agile development is “well suited to the ex-

$^k$https://github.com/tseemann/snippy
$^l$https://www.r-project.org/
The exploratory, iterative and collaborative nature of scientific inquiry” (Kane et al. 2006).

In the case of Walsh et al. (2016a), Mac Aogáin et al. (2017) Walsh et al. (2017a) and Palu et al. (2019), there was a particular reliance on this dynamic. For reasons of complexity in the nature of the pipelines, uncertainty around the intended outcomes, and the tentative nature of the proposed solutions (using interim results to choose next steps), numerous iterations of the pipeline software were needed. The importance of “striking while the iron is hot” is important both for the software engineer and the bioinformatician, as both attempt to understand the latest results and redirect their efforts. The bioinformatician must re-evaluate the accuracy and correctness of the pipeline from a theoretical point of view. The software engineer must be ready to re-implement parts of the pipeline’s code. In both cases, a fresh “mental model” of the current solution from theoretical and implementation perspective is important. If too much time is allowed to pass between iterations, important, non-documented contextual information can be lost or diluted, and must be re-constructed when work is recommenced. Iterations must therefore be made quickly - but also systematically. Agility must be balanced with process and organisation. It must always be possible to reverse out of blind alleys, and to review the steps that have led to them. In addition, it must be possible to reproduce the running pipeline at will.

In these four distilled papers in particular, the use of the Docker tool as the shared platform for iterative collaboration, and the Dockerfile as the means of exchange of pipeline updates (supported by a reliable source control system) permitted this combination of agility and reliability.

By taking Walsh et al. (2017a) as an example (see figure 6.2 for a visual representation of the pipeline), we can see how this dynamic played out in reality.

The premise of the research was that by curating a database of organisms, genes and antibiotics, - a “resistance profile” - it becomes possible to “call” resistances of sample reads.
For the purpose of this thesis, the actual bioinformatic details are secondary. They are presented here to give context. The scope of the following description of pipeline development is to explain the methods by which a software engineer can collaborate with a bioinformatician in a way that respects the opposing requirements of agility and reliability.

Key to a successful collaboration is a clear separation of concerns. In this case, the database, which was a comma-separated file, was exclusively maintained by the bioinformatician. In addition, the design of the pipeline (the choice of tools, their order and their parameter values) was also the bioinformatician’s responsibility.

Where custom elements were required in the pipeline, this work was done by the software engineer, using Perl, based on high-level requirements created by the bioinformatician. The preparation and installation of the tools (custom and standard) was undertaken by the software engineer, and codified as a Dockerfile. All code, including the database file, custom perl and Dockerfile, was committed to a single source code repository.

Development proceeded by iterative improvements and extensions to the pipeline, as it moved from parsing the database, to invoking tools to build reference index files, perform alignments, make variant calls and then interpret those calls into resistance calls. Once the tool was able to first make some predictions, the bioinformatician was able to measure the sensitivity and specificity of those results and identify errors or limitations either in the theoretical design, or the implementation details of the pipeline. Depending on where subsequent changes were needed, either the bioinformatician or the software engineer would commit those changes to the source control, and a new Docker image would be built from the source and tested.

This cycle continued until results meriting publication were achieved.

Note that the TB pipeline was extended for use with C. difficile by adding C. difficile elements to the database and branching the pipeline to account for acquired genes as well as inherited ones. The evolution to the code was managed in the
same kind of iterative way as the original development. The exchanges between the bioinformatician and the software engineer during this time were a good example of how code, at the right level of abstraction, can be the *lingua franca* between the two disciplines. Although we were discussing biological questions (for example the difference between chromosomal genes and plasmids in the context of resistance to antibiotics) these questions were couched in terms of the data and the code that manipulated it.

In the case of Palu et al. (2019), the process was similar in form, but not identical in the details. Much of the code for this publication was custom R code, rather than a pipeline of standard tools. The process was still iterative and the roles were still separated along designated lines, with Docker again being the technology at the boundary. The bioinformatician took responsibility for the R code development and testing within the context of a Docker environment created and maintained by the software engineer. Many iterations required no input from the software engineer, and where they did, they tended to involve tasks like installing corrected versions of required Linux and R packages.
Figure 6.2: Resistance Prediction Pipeline

Resistome DB

DB Parser

Genes Parser

Ref Genes target

bowtie-build

Bowtie DB

bowtie2

Ref Genes.sam

samtools view -b -S /f-s

Map Genes.bam

samtools sort

Map Genes.bam

samtools mpileup -uf

bcftools call -c

snpseq.

seqsect A

Parse VCF File

Resistance Calls

Profile

"Dictionary" of resistances, indexed on organism, gene and drug.

Acquired Genes target

bwa index

bwa index files

bwa mem -b

bwa.sam

bwa.bam

samtools sort

bwa tmp.bam

samtools index

bwa output bam

samtools depth

bwa output depth

Parse Depth File

Looks for matches with organism and gene and checks for variant match

Looks for depth/coverage thresholds for organism and gene

Data

Process

Simple
6.4. Discussion

6.4.1. The Multidisciplinary Team

The team, as mentioned above, was a multidisciplinary one, composed principally of software developers and bioinformaticians. This is different to most software scrum teams and presented a challenge when it came to work allocation. Not all work items could be carried out by all team members. An example of this is the fact that the software engineers were not typically able to define meaningful systems tests for new functionality. Similarly, the bioinformaticians did not necessarily have familiarity with the latest software techniques and technologies that some issues required. The problem was mitigated during the planning meeting for new sprints, by balancing the issues chosen for the new sprint between these two groups.

Another difference between this multidisciplinary team and common industry software teams was the variety of tasks required of the bioinformaticians. In the case of this organisation, it was expected that the bioinformaticians would produce research-driven academic publications, and take part in academic conferences. This meant that they were not available as full-time resources during a sprint. This problem was mitigated by creating a separate project to manage Research and Development issues. This project ran continuously in the background and was used by bioinformaticians to record progress on this aspect of their work. The planning meeting also helped to reduce the disruption caused by part-time resourcing: At the start of every sprint each contributor would specify the percentage of their time that they were able to dedicate to the project for that sprint. This allowed us to calculate the total sprint capacity.

6.4.2. Bioinformatician as SME

Over a number of sprints, some changes to the process were made. It was important to correctly identify the roles of scientists in the scrum setting. For example a scien-
tist/bioinformatician is a Subject Matter Expert (SME) or a product owner. They are consulted during a sprint, or called on to provide detailed specifications even in code if necessary. They are not necessarily responsible for developing production-ready code. Another important role for bioinformaticians was as testers and reviewers of the biological conclusions of pipeline output.

6.4.3. Observations

A number of observations can be made about the way in which the bioinformaticians were integrated into this software development process. In each case, one can talk of integration points between bioinformatic concerns and software development concerns that were provided by the process structure and toolset. These integration points acted both to provide the bioinformatician with the advantages of modern software development but also to protect them from some of its unnecessary (for them) complexity.

Docker as Integration Point

The use of Docker images within the project was sophisticated. Each pipeline has three Dockerfiles, one layered on the other in the way described in figure 2.7 in section 2.4.1. The lowest one, called base, defined the basic Linux distribution, and the non-bioinformatic tools needed (e.g. curl, java, python). The next layer up, called tools, defined the bioinformatic tools needed by the pipeline, for example qiime or samtools mentioned previously. Finally, the last layer was where the custom code of the pipeline itself was installed. The reason for this subdivision was to facilitate quicker builds of the Docker images, and make it possible to re-use Dockerfiles in later pipelines. Maintaining such Docker images added to the administration overhead of the pipelines, but for sound engineering reasons.

The bioinformaticians did not need to concern themselves with these details. The software developers created and maintained the Dockerfile definitions for the images,
and made them available to the bioinformaticians. This reduced the cognitive load on the bioinformaticians to that of knowing how to run Docker images as containers and it protected them from the underlying details.

In this way, Docker acted as an integration point between the two domains, demarcating roles and at the same time, providing a service.

**Git as Integration Point**

In section 6.2.3, the technique of feature branching was outlined, and its purpose explained. While the concept is straightforward the concrete steps to creating and merging these branches requires training and practice, and a well-developed internal model of the underlying tool - Git.

If the responsibility for the creation and merging of feature branches is assigned to the software engineer, then the bioinformatician need only know how to *check out* such branches, and commit work to them, in situations where their specialised coding input was required. Moreover, the bioinformatician need never be concerned with the creation of release branches, or the tagging of release versions.

Feature branches become the integration point and demarcation zone in the same way that Docker images do. They constitute the point at which responsibility crosses over from one role to another, and they maintain the right level of abstraction for the bioinformatician, bringing them into a more rigorous process, without creating excessive cognitive burden.

**Jira as Integration Point**

Finally, and in a similar way to Docker and Git previously, the use of Jira created a clean point of interface between the roles of software developer and bioinformatician.

In their capacities as SMEs and as specialised developers, Jira was the common reference point. It was where work was allocated, and where initial requirements and subsequent conversations about such work could be captured.
Jira has a friendly and intuitive user interface, and it models concepts that are also intuitive - issues to be assigned, read and worked to completion. The bioinformaticians did not need to be shielded from its complexity in the same way that Docker and Git required. It became, as in any software project, the interface where information was exchanged transparently across various roles in the organisation.

6.4.4. Limitations of Reproducibility

Taking a step back to see the ways in which software engineering influences have made reproducibility more achievable, we can detect two major themes: Abstraction and Standardisation. The deployment targets for which we create software have been steadily changed from specific hardware platforms, to more abstract virtual environments. These environments have become de facto standards through competition and widespread acceptance. The source control and build systems - including their dependency management systems with public library repositories like Maven - have also become industry standards.

The problem of reproducibility hasn’t gone away. Instead, it has been rendered less difficult by increased uniformity in tools, techniques and platforms. But this also indicates the limitations of these solutions. Standards change. Although Java has been around for 20 years, the advent of containerization makes its JVM less relevant. Although Maven has displaced Ant and Make as the best-known industry build system, it may fall into disuse as rivals (like Gradle) gain more traction. Even if Kubernetes remains a deployment standard for the next 20 years, it will evolve and grow and old versions will no longer be supported on public clouds (this has already happened once for the Apache Kafka experiment).

These limitations do not render the approaches described in this chapter pointless. Stated more completely, the object of the exercise is to maintain reproducibility for as long as possible or necessary. The limitations can be addressed in part by using the latest versions available, and adopting a maintenance phase for key software
projects and experiments.
7. The Roles of Code

A version of this chapter has been presented as the paper *The Roles of Code in Biology* which has been published in *Science Progress* (Sage).

The paper presents a novel taxonomy of the different ways in which code is employed, and through the lens of this taxonomy, provides a software engineering perspective on ways in which biologists and bioinformaticians could more productively engage with code and with software engineers. The taxonomy is in fact applicable to any scientific programming context.

- **Views**: n/a
- **Altmetric**: 9

### 7.1. The Future of Software in Bioinformatics

Bioinformatics began as a novel perspective on biology: viewing biological processes as information systems. It integrated information theory and computational thinking with the biological domain. Over time, the word *bioinformatics* came to mean
something else - the use of software to process large stores of biological (often genomic) data. This definition reflects the current state of the art - an integration of the biological domain with computer programming and its related techniques, rather than with computational science.

Some observers, including the person credited with coining the term originally, believe the older definition of bioinformatics will return (Hogeweg 2011). Whatever happens, scientific progress in biology will continue to be inextricably bound to its effective use of software (Storer 2017), and this holds both opportunities and risks for the research community.

The risks lie in the fact that according to the literature reviewed in this thesis, and indeed according to direct findings of this research, scientific programming is typically not done in a way that promotes reproducibility or scalability.

On the other hand, there are new trends in professional software engineering which are architectural and process-based in nature, and which address both of these aspects of software development which can be incorporated into bioinformatics, as presented in chapters 4, 5 and 6. One can say that the practice of software engineering has undergone a great deal of evolution in the recent past. A number of important architectures, frameworks and paradigms have emerged in the last decade or so to facilitate the creation of powerful and complex systems with relatively few resources. The actor architecture described in chapters 4 and 5 (and the co-routines and continuations that have emerged since), are examples. New de facto standards, and their supporting frameworks, make it easier to create and process streams of large datasets, including the transparent handling of back pressure. A return to functional programming, heralded by the arrival of a number of new programming languages such as Scala and Kotlin, and supported by recent changes in existing languages such as Java, has underpinned the industry’s ability to create parallel and distributed systems of ever increasing scale. Containerisation has allowed developers to more easily isolate and reproduce their work. Cloud-native architectures and
frameworks, the wider use of Infrastructure as Code, and cloud-based operating systems such as Kubernetes simplify the creation and maintenance of sophisticated hardware configurations on premises and in the cloud.

What have all these advances in common? They provide abstractions over specialist software engineering concerns, allowing small teams of generalists to build complex, reliable distributed systems. They democratise the field by widening access to skills and resources that previously would only have been available to large institutions.

While it is a productive time to be a software engineer, one might say that it remains a difficult time to be a bioinformatician. The field has made huge strides, and has taken clinical and research biology in the direction it needs to go. But the day-to-day experience of bioinformaticians is often an exercise in frustration with hard-to-use tools and limitations of scale with regards to data and processing power. Moreover, a crisis in publishing reproducible research, evidenced in the literature, is being felt in the lab. There is a gap between the ambitions and abilities of this relatively young discipline.

However, this gap can be bridged. It has happened in one generation of software development, and it can happen much faster in bioinformatics if the lessons learned in the one field can be directly applied to the other (Fauilk et al. 2009).

This thesis has listed some of the challenges being faced by the current generation of bioinformatic researchers, and it has presented a number of examples of point-solutions to these problems, through the application of software engineering best practices and latest technologies. A more general conclusion can be drawn - that introducing such point-solutions will require the sustained and strategic application of software engineering expertise to the field of bioinformatics. Rather than expecting bioinformaticians to build, execute and maintain tools of ever-increasing scale and sophistication in a reproducible manner, leaders in clinical and research bioinformatics should consider directing more of their resources towards the professional
construction of such tools.

Bioinformaticians aim at understanding biological questions, often through the gathering and analysis of large amounts of biological data. But they consider the software aspects to be secondary, as a kind of "tax" to be paid in order to do their work (Wilson 2014), and certainly not a "first order concern" (Prabhu et al. 2011).

This is natural and correct. Today, almost all undertakings, be they commercial, political or scientific, have a dependency on software. That does not make such undertakings cross-disciplinary, and the SMEs in each case do not consider the development of software to be their primary skill or interest. Instead, they have come to understand the importance of interacting with software engineers, thereby leveraging decades of research and development in that field to support progress in their own.

The literature review looked at the history of bioinformatics, and found that it has changed its emphasis over time. In the context of the problems that currently challenge the field, it may be time for bioinformatics to examine its mission and to adjust its approach accordingly. A first step would be to redefine its core competencies, and re-designate software engineering as a vital - but peripheral - service, separate from but complementary to, the computational and data sciences at the core of bioinformatics.

The idea of software engineering as a service to bioinformatics may find resistance in some quarters, given past controversies over the role of software code as presented in chapter 2. But these misgivings can be addressed by the realisation that code plays multiple roles in applied software engineering.

Presented here are three different dimensions (abstraction, subdomain, communication) along which these purposes vary, which can be mapped to the biological field. The taxonomy leans heavily on software engineering research, but presents it in a way that provides insights to a life science audience, and indicates optimal ways to develop software in biological and bioinformatic research contexts.
With this more refined consideration of the various roles of software in biology, some implications for the relationship of software engineering as a discipline to the life sciences are discussed. The purpose of examining this relationship is to address its role in solving current issues in reproducibility, scalability and productivity in bioinformatic software.

7.2. Abstraction, Subdomain, Communication

In the absence of any literature on how to classify the different roles of code in software engineering, a novel taxonomy is presented here based on direct experience of that discipline, and with influence from multiple software engineering sources. The aim of this approach is to describe current (often informal) software practices to a non-software engineering audience, in a more formalised way that provides useful insights.

Firstly, rather than a hierarchy of value, where elements at the top are of greater worth than those underneath, in software there exists a stack, a standing-upon-shoulders, where higher levels of abstraction derive their power from the expert implementation of the levels below them. Code itself can function on many levels, from the assembler of device drivers, to cloud-native infrastructure-as-code, library code, domain-driven designed application code, and custom-built domain-specific languages. None is of any greater intrinsic worth than any other. To quote Liskov & Guttag (1986), abstraction serves to hide “‘irrelevant’ details, describing only those details that are relevant to the problem at hand”. While abstraction is a continuum, it can be useful to break out some discrete values. In the context of scalable cloud-native systems, the following values are proposed for the abstraction dimension of the taxonomy:

- **System**: Uses system languages like assembler, C, C++ and Rust to specify generic software abstractions like bytes, strings, arrays etc. or concepts close
to the specific hardware like Graphical Processing Units (GPUs) or Single Instruction Multiple Data (SIMD).

- **Infrastructure**: Specifies hardware and middleware configurations, particularly on cloud infrastructure, using languages like Terraform or vendor-specific representations like CloudFormation.

- **Application**: Creates and uses abstractions that map to real-world concepts from the field to which the software is being applied, using languages like Java, C#, Python, R etc. Contains primary so-called *business logic* - the knowledge of the problem domain, and its solutions.

- **Orchestration**: Specifications of how to deploy and coordinate multiple application-level programs on infrastructure, in particular cloud, using markup languages like YAML and a variety of platforms like Docker and Kubernetes.

In operational software systems, the level of abstraction of a layer of code can be usefully thought of as the the order in which it is applied when building that system.

Secondly, when crafting a software system, there are distinct problem areas that one must address. The widely-followed software engineering technique *Domain Driven Design* (Evans 2004) calls these areas *subdomains* and categorises them into Core, Supporting and Generic types.

- **Core** subdomains address the central mission and distinguishing feature of the system.

- **Supporting** subdomains perform mission-specific work that is needed by the core.

- **Generic** subdomains provide utilities that are needed by the system but are not specific to it.

Note that code at any layer of abstraction can belong to *core, supporting* or *generic* contexts. For example, code that uses GPUs to achieve high degrees of
parallelization in modelling intra-cellular processes would be considered to have a *system* level of abstraction while being a *core* subdomain. However some Kotlin code created to report experimental results in an interactive table on a browser would be at an *application* level of abstraction and in a *supporting* subdomain.

Finally, software engineers use code not only as a means of directing a computer to accomplish a given task, but also as a way to communicate a problem, and its solution, in an unambiguous way. To quote Martin Fowler, “Any fool can write code that a computer can understand. Good programmers write code that humans can understand” (Fowler 2018). Good code, he suggests, should be intelligible to other humans, not just computers, so that other humans might understand how a problem is framed, and how its solution is fashioned, so as to more easily understand the problem, fix or improve the solution, and know how to adapt it to solve other related problems. Code can be, in other words, a way of communicating complex concepts between humans, in a rigorous and reusable fashion. Whether it is used in that role depends on the way it is written, the language used, and the costs involved in writing code this way.

- **Machine**: Code which only the computer can read.

- **Human**: Code which explains its intent to the human reader as well as the target computer.

Note that while *system* level code will tend by its nature to communicate principally to the machine because it typically deals in concepts closer to the hardware, a separate dimension of *communication* still makes sense because code at higher levels of abstraction is free to be written in a way that communicates either with machine or human.

The following table summarises the proposed taxonomy of the roles of code.
Table 7.1.: Role dimensions and their values

<table>
<thead>
<tr>
<th>Role Type</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstraction</td>
<td>System, Infrastructure, Application, Orchestration</td>
</tr>
<tr>
<td>Subdomain</td>
<td>Core, Supporting, Generic</td>
</tr>
<tr>
<td>Communication</td>
<td>Machine, Human</td>
</tr>
</tbody>
</table>

Note that whether it is a good idea or not to implement core domains using system level abstractions which perhaps do not communicate well to a human reader, is an interesting question but not strictly in the scope of this thesis. However framing the question in terms of this taxonomy can be a useful way to approach the problem.

7.3. Mapping Code to Biology

We can map this deeper understanding of the nature of software development onto the domain of biology in such a way as to cast light on the different roles of code in the life sciences. Perhaps in doing so, some of the remaining heat can be removed from the past controversies referred to in the literature review of chapter 2, specific directions for life science researchers with respect to how they use code can be suggested, and ways in which to move in those directions can be indicated.

Some biological problems are computational in their very nature, or are more amenable to computational representation. In fact the term bioinformatics was first used in order to highlight such “informatic processes in biotic systems” (Hogeweg 2011). Examples might include the mathematical analysis of physiological processes such as the seminal paper on neuron spiking by Hodgkin & Huxley (1952) which actually predates the term bioinformatics, or more recently the practice of whole-cell modelling (Karr et al. 2015). Code that models such problems would be considered as belonging to the core subdomain type, whatever the level of abstraction used to implement it, or to whatever extent the code communicated to a human audience.

In the years since the term bioinformatics was coined, it has come to mean the use
of software in the processing and management of biological data, often at high scale. In this context, bioinformatics code is likely to be found in all subdomain types: core, supporting and generic, and across all levels of abstraction from system to orchestration, and with the purpose of communicating both to human and machine.

The roles of code in biology, then, consist of modelling biological elements that are intrinsically computational, and to engineer scalable and reproducible solutions to process vast and growing amounts of biological data.

But code which models computational processes in biology is no more important than code which distributes biological data, and its processing pipelines, across clusters of servers and processors. Each includes elements that are core to the mission, support it, or are generic. Each can include code that operates from system level to orchestration level. This is not a hierarchy of worth, but the same stacked layering of abstraction that applies in any application of software. Past debates over the importance of code in the life-sciences are off-the-point and wasteful. Code is neither central nor peripheral. It is both.

Moreover, a realisation that there are many roles of code in biology can steer the conversation in a more useful direction: who should fulfil which roles? As figure 7.1 indicates, when it comes to the life sciences, there is a “sweet spot” on which the three dimensions of role converge. This is the point where biologists and bioinformaticians can derive the most value from code. This is where code addresses core questions of the domain, at a real-world level of abstraction, and communicates its intent to human readers. The more life scientists can move their coding towards this sweet spot, the more their code will serve them, and the less they will have to serve their code.

It is useful to think of the sweet spot as containing the greatest concentration of essential complexity, from the biological perspective, while the others contain mostly accidental complexity, to use the terminology from Brooks (1987).

The other roles can be left to software engineers, by using outsourced products
Figure 7.1.: Sweet spot for life scientists and bioinformaticians

The sweet spot is where bioinformaticians get the best return on their efforts. The three dimensions of code roles converge and services, and by integrating software engineers into the research team. To enable this kind of integration, some attention must be given to team structure. According to Conway’s Law, “there is a very close relationship between the structure of a system and the structure of the organisation which designed it” (Conway 1968). The structure of a research organisation can either be an asset or a liability when it comes to the way it designs software systems, by either enabling or preventing designs that permit scientists to work in their “sweet spot”, while engineers work in theirs.

Using what has become known as the “inverse Conway manoeuvre” (Skelton & Pais 2019) organisations can restructure to promote more suitable designs outcomes. In order to move life science researchers (i) towards core subdomains and away from supporting subdomains, and (ii) towards application abstractions and away from system, infrastructure or orchestration abstractions, such restructuring would involve filling the void left behind with software engineers. Code at the boundaries between these competencies would then need to fulfil the human communication role. This research has already led to suggested ways in which this can be done (Lawlor & Sleator 2020). Interactions at team interfaces will be made smoother by scientists following existing advice on good programming practices (Taschuk & Wilson 2017), and by software engineers adapting to the needs of scientific computing.
and providing scientifically useful abstractions over computing systems (Faulk et al. 2009).

7.4. Case Study

Examined through the prism of the taxonomy, the work presented in the case study from chapter 6 falls into the following categories:

- **Infrastructure, Supporting, Human**: Creating Docker\(^a\) images to support the code dependencies of pipelines, using descriptive Dockerfiles as the code.

- **Orchestration, Core, Human**: Coding deployment descriptors to define the deployment environment of pipelines running in Docker images. This was done at the time using Docker Swarm and Docker Compose, but today would be done using Kubernetes\(^b\).

- **Infrastructure, Generic, Human**: Creating cloud-native job queuing system to permit pipeline containers to pick jobs and work on them in a scaled environment, using Java code and following software best practices for modularity and readability.

- **Application, Supporting, Human**: Creating a standardized pipeline Application Programming Interface (API) to read queued jobs, invoke bioinformatic pipelines based on the job content, and extract the pipeline results, using Java code and following software best practices for modularity and readability.

- **Application, Core, Human**: Developing bioinformatic pipelines using Python and R, conforming to the standardized API mentioned above, and designed to run in the dedicated Docker containers mentioned above.

\(^a\)https://www.docker.com
\(^b\)https://kubernetes.io
The bioinformaticians on the team worked exclusively in the final category, corresponding to our identified *sweet spot*. All other roles were filled by software engineers. This meant that the bioinformaticians did not need to concern themselves with matters of scale. Where increased load was needed, multiple Docker images were configured to run in parallel (on cloud infrastructure if necessary) and chaining of the results of one pipeline into the input of another was handled by *Simplicity*. In addition, the resulting pipelines were automatically reproducible, because all their runtime dependencies were encapsulated within Docker containers.

This approach was made possible by a system architecture which mirrored the communication structure of the organisation. The software engineers were divided into teams that worked on user interfaces, and teams that worked on the backend infrastructure. Each pipeline had its own lead bioinformatician, who developed their pipelines to a specific common API. The bioinformaticians communicated with the infrastructure software engineers, using language grounded in terms of that API. They communicated with the PI using language of the core (biological) subdomain.

### 7.5. Discussion

One difficulty to be addressed as part of this proposed approach is hinted at by (Prabhu et al. 2011) when they quote one scientist as saying that even “funding agencies think software development is free”, and regard development of robust scientific code as “second class” compared to other scientific achievements. The way in which research projects are funded does not currently take into account the costs associated with developing software. This is echoed by Faulk et al. (2009) who point out that “[p]urchasing decisions and budgeting models are short-term and hardware-focused”, with little regard for software.

While not every project will be able to budget for a full-time software engineer, research groups should be able to share such resources, or make use of specialised
external software companies which would grow in number to meet demand. In her paper on *big biology*, Vermeulen (2016) notes the movement in biological research to centralize complex and expensive technologies (e.g. “electron-microscopy, NMR spectroscopy, röntgendedefraction, ultracentrifuge”) with the goal of “not only the sharing of costs, but also the development of professional operational skills”. In the same context of *big biology*, this sharing of costs and skills should be applied to software engineering. This would call for an explicit position for software engineering within bioinformatics.

This should be seen as a natural progression in a maturing field of study. The first scientific astronomers fashioned their own telescopes. Seminal work in electromagnetism was performed on improvised experimental equipment. But today’s Hubble telescopes and Large Hadron Colliders are the fruit of collaboration between researchers and service providers.

The presence of a software engineer does not in any sense undermine the cross-disciplinary nature of bioinformatics. On the contrary - each has the skills and vocabulary to communicate with the other, while maintaining separate roles and functions as demonstrated in the case study of chapter 6 (Lawlor & Sleator 2020).

The bioinformatician will very often communicate with the engineer using source code. As suggested by (Wilson 2006b) it would be best if the bioinformatician also had a working knowledge of the basic tools of software engineering such as source control and unit tests. Some amount of coding will always be an essential part of the bioinformatician’s toolkit but this code should be seen as a point of overlap between that field and software engineering rather than a research output, if questions of scale and reproducibility are to be effectively addressed.

The assertion that “biologists are all bioinformaticians now” (Stein 2008) is true in the same sense that every business is a technology business. But we don’t expect SMEs to train in “requirements engineering” - that’s what software engineers are for.
It is a contention of this thesis that the natural way to introduce these software engineering tools and techniques would be to incorporate software engineers themselves, as service providers, whose mission it becomes to democratise bioinformatic research through the development of suitable bioinformatic abstractions over reliable, scalable and reproducible tools. This would have a comparable effect on bioinformatics that the democratisation of cloud computing is having on software development: allowing small and medium-sized teams to “punch above their weight”.

By explicitly recognising the difference between the data and computer sciences on one hand, and software engineering on the other, bioinformatics can be liberated from software-related toil outside of its core domains, and concentrate instead on answering biological questions.

7.6. Further Research

Taking stock of the findings presented in this thesis, one can divide it into two groups. Firstly, the first-author work has consisted in publishing around the use of innovative software engineering techniques and tools in order to improve outcomes in bioinformatics. Secondly, as a contributing author, a number of bioinformatic applications of these tools and techniques have been presented. Collectively they have rendered this bioinformatic work more reproducible, and have demonstrated ways of permitting effective and rapid collaboration between bioinformaticians and software engineers, thus placing cutting edge software engineering tools and techniques at the service of biological research.

The intention in future work is to bring these two strands closer together. One planned line of inquiry, for example, is to combine the work done related to scaled computation (Lawlor & Walsh 2016) and that related to scaled data (Lawlor et al. 2018), and bring this to bear on questions of microbial resistance (Walsh et al. 2018).
The bioinformatics community needs a data backbone that is better suited to the scale and nature of the data being processed and above all that is more amenable to parallel processing. Chapters 4 and 5 present jigsaw pieces of a larger puzzle, and more work is needed to demonstrate more efficient alternatives to the construction of bioinformatic pipelines.

Another area worthy of study is the application of DSLs to the creation of bioinformatic pipelines. As indicated in Lawlor & Sleator (2020), finding the right level of abstraction for bioinformaticians to work at will be crucial in freeing them to be more productive. Functional programming languages are particularly adapted to implementing DSLs (Hu et al. 2015), so this line of inquiry follows on naturally from the work done using Scala. Since beginning this research, another programming language, Kotlin (Samuel & Bocutiu 2017), has broken into the top tier of general purpose programming languages used in the software industry. Kotlin permits a functional style of programming, and so should be considered a suitable candidate for use in Bioinformatics, for similar reasons to those outlined in section 2.4.2 with respect to Scala. Internal (also known as embedded) DSL implementations such as those available through Kotlin (Considine et al. 2019) and Scala (Barringer & Havelund 2011), could offer the right balance between control and simplicity, and the right level of abstraction, for bioinformaticians to construct complex pipelines from disparate tools and components.

It is worth pointing out another potential continuity in this research that the use of Kotlin would bring about. Kotlin supports the co-routines programming paradigm (Conway 1963), which can be seen as an alternative interpretation of the actor model (Shaver & Lee 2012), presented in chapter 2 and used extensively in this research (Lawlor & Walsh 2016, Lawlor et al. 2018).
Principal Conclusions

- There are issues of scalability and reproducibility in the use of software in bioinformatics, which as a discipline is lacking in software engineering processes and architectures.

- An intelligent integration of software engineers and bioinformaticians will address these issues and gaps.

- Actor architecture as implemented by the Akka library, coupled with Reactive Streams and Functional Programming languages provides an alternative that outperforms existing solutions for scalability in Bioinformatics.

- The Distributed Log abstract data structure, as implemented by Apache Kafka, offers a promising format for primary bioinformatic data repositories, as it lends itself to a high degrees of parallelization, distribution and reliability.

- Design aspects of Apache Kafka and Akka actors present opportunities to create ecosystems of distributed data and distributed computation, in which data can be continuously updated and continuously processed.

- Based on a case study, in which a series of bioinformatic pipelines were created by bioinformaticians in collaboration with software engineers, the integration of these two disciplines can be seen to be not only possible but productive.

- Containerisation, as implemented by Docker, offers an engineering solution to bioinformatic reproducibility, and serves as a handover mechanism as part of a collaborative model between software engineers and bioinformaticians.

- The roles of code itself, especially in scientific programming contexts such as bioinformatics, are varied and nuanced. A taxonomy is presented to guide bioinformaticians towards a sweet spot in those roles, where their use of code is likely to result in the best results.
Bibliography


Bain, S. (2008), Emergent design: the evolutionary nature of professional software development, Pearson Education.


URL: https://doi.org/10.1093/biosci/bix034


Ioannidis, J. P. (2005), ‘Why most published research findings are false’, *PLoS medicine* 2(8), e124.


Lawlor, B. & Walsh, P. (2016), The weekend warrior: how to build a genomic supercomputer in your spare time using streams and actors in scala, in ‘Ubiquitous Intelligence & Computing, Advanced and Trusted Computing, Scalable Computing and Communications, Cloud and Big Data Computing, Internet of People, and

URL: http://www.tandfonline.com/doi/abs/10.1080/14636778.2013.773172


URL: http://dl.acm.org/citation.cfm?id=333067.333074

Morris, C. (2008), Some lessons learned reviewing scientific code, in ‘Proc 30th Intl Conference Software Eng (iCSE08)’.


URL: [http://dx.plos.org/10.1371/journal.pcbi.1002487](http://dx.plos.org/10.1371/journal.pcbi.1002487)


Popper, K. (1935), ‘The logic of scientific discovery’.


**URL:** [http://rgdoi.net/10.13140/RG.2.2.16070.88646](http://rgdoi.net/10.13140/RG.2.2.16070.88646)


B. Scala Code Extracts

B.1. Code Snippets

Listing B.1: QueryBatchWorker Actor

```
class QueryBatchWorker extends Worker[QueryBatch] {
  implicit val sys = context.system
  implicit val mat = ActorMaterializer()
  val settings = Settings(context.system)
  val databaseDirectory = settings.DatabaseDirectory

  CreateFastaFiles.createFastaFiles(settings.DatabaseDirectory, 10, 1000, 3000)

  def databaseNames(): Seq[String] =
    (1 to 10) map { i => s"database_$i" }

  override def doWork(work: QueryBatch) {
    try {
      val queryBatchCollector = context.actorOf(
        QueryBatchCollector.props(
          work.queries.length,
```
work.replyto,
self))

Source.actorPublisher(QueryPublisher.props(work.queries)).
runWith(Sink.actorSubscriber(
QuerySubscriber.props(queryBatchCollector,
work.queries.length)))

} catch {
  case e: Exception => log.error(e, "Ooops")
}

Listing B.2: Query Subscriber Actor

class QuerySubscriber(batchCollector: ActorRef,
  numberOfQueries: Int)
  extends ActorSubscriber with ActorLogging {
    override protected def requestStrategy: RequestStrategy =
      new MaxInFlightRequestStrategy(10) {
        override def inFlightInternally: Int = numberOfQueriesOpen
      }
    implicit val sys = context.system
    implicit val mat = ActorMaterializer()
    var numberOfQueriesOpen = 0
    var numberOfQueriesCompleted = 0
    var numberOfCompletedQueryCollectors = 0
    def totalQueries =
      numberOfQueriesCompleted + numberOfQueriesOpen
def receive: Receive = {
    case OnNext(query: Query) =>
        log.error("Got a new query {}", query.id)
        val queryCollector = context.actorOf(
            QueryCollector.props(numberOfQueries,
                batchCollector))
        context.watch(queryCollector)
        Source.actorPublisher(
            DatabasePublisher.props(query.databases)).
            runWith(Sink.actorSubscriber(
                DatabaseSubscriber.props(query,
                    queryCollector)))
        numberOfQueriesOpen += 1
    case OnComplete
        if numberOfCompletedQueryCollectors == totalQueries =>
            context.stop(self)
    case OnComplete =>
        context.become(incomingStreamCompleted)
    case QueryComplete =>
        numberOfQueriesOpen -= 1
        numberOfQueriesCompleted += 1
    case Terminated(_)
        if numberOfCompletedQueryCollectors == totalQueries - 1 =>
            context.stop(self)
    case Terminated(_) =>
        numberOfCompletedQueryCollectors += 1
    case whatever =>
        log.error("Missed {}", whatever)
}

def incomingStreamCompleted: Receive = {

case QueryComplete =>
  numberOfQueriesOpen -= 1
  numberOfQueriesCompleted += 1

case Terminated(_)
  if numberOfCompletedQueryCollectors == totalQueries - 1 =>
    context.stop(self)

case Terminated(_) =>
  numberOfCompletedQueryCollectors += 1

}

Listing B.3: Database Subscriber Actor

class DatabaseSubscriber(query: Query,
  queryCollector: ActorRef)
extends ActorSubscriber with ActorLogging {
  implicit val sys = context.system
  implicit val mat = ActorMaterializer()

  override protected def requestStrategy: RequestStrategy =
    new MaxInFlightRequestStrategy(10) {
      override def inFlightInternally: Int = numberOfDatabasesOpen
    }

  var numberOfDatabasesOpen = 0
  var numberOfDatabasesClosed = 0
  def totalNumberOfDatabases =
    numberOfDatabasesOpen + numberOfDatabasesClosed

  def receive: Receive = {
    case OnNext(database: Database) =>


try {
    log.info("Got a new database for query", database.name, query.id)
    Source.actorPublisher(
        FastaFilePublisher.props(database.name)).
    runWith(Sink.actorSubscriber(
        DatabaseSequenceSubscriber.props(query, database.name, queryCollector)))
} catch {
    case e: Exception => log.error(e, "Error creating database sequence subscriber")
}

numberOfDatabasesOpen += 1

case OnComplete
    if totalNumberOfDatabases == numberOfDatabasesClosed =>
        context.parent ! QueryComplete
        context.become(incomingStreamComplete)

case OnComplete =>
    context.parent ! QueryComplete
    context.stop(self)

case OnError(err: Exception) =>
    log.error(err, "Error in database stream.")
    context.parent ! QueryComplete

case DatabaseComplete =>
    numberOfDatabasesOpen -= 1
    numberOfDatabasesClosed += 1

case whatever => log.error("Missed", whatever)
}
Listing B.4: Database Sequence Subscriber Actor

def incomingStreamComplete : Receive = {
    case DatabaseComplete if numberOfDatabasesOpen == 1 =>
        context.stop(self)
    case DatabaseComplete =>
        numberOfDatabasesOpen -= 1
        numberOfDatabasesClosed += 1
}

class DatabaseSequenceSubscriber(query: Query,
                                  databaseName: String,
                                  queryCollector: ActorRef)
extends ActorSubscriber with ActorLogging {
    override protected def requestStrategy: RequestStrategy =
        new MaxInFlightRequestStrategy(50) {
            override def inFlightInternally: Int = incompleteCalcs
        }

    var incompleteCalcs = 0
    def receive = init

    val collector = context.actorOf(
        QuerySingleDatabaseCollector.props(queryCollector))
    context.watch(collector)

    def init: Receive = {

```scala
case OnNext(StopSequence) =>
  collector ! DatabaseSize(0)

case OnNext(databaseSequence: DatabaseSequence) =>
  val alignmentWorker = context.actorOf(
    AlignmentActor.props(collector))
  alignmentWorker ! Align(query.sequence, 
    databaseSequence, 
    query.scoreMatrix)
  context.become(runningQuery(1))
  incompleteCalcs = incompleteCalcs + 1
...

def runningQuery(noOfQueriesInFlight: Int): Receive = {
  case OnNext(StopSequence) =>
    collector ! DatabaseSize(noOfQueriesInFlight)
  case OnNext(databaseSequence: DatabaseSequence) =>
    val alignmentWorker = context.actorOf(
      AlignmentActor.props(collector))
    alignmentWorker ! Align(query.sequence, 
      databaseSequence, 
      query.scoreMatrix)
    context.become(runningQuery(noOfQueriesInFlight + 1))
    incompleteCalcs = incompleteCalcs + 1
...
  case OnComplete =>
    context.parent ! DatabaseComplete
    log.info("Stream is complete with \{\} queries in total", 
      noOfQueriesInFlight)
    collector ! DatabaseSize(noOfQueriesInFlight)
```
case Terminated('collector') =>
  context.stop(self)
case AlignmentComplete =>
  incompleteCalcs = incompleteCalcs - 1
case whatever => log.error("Missed{}, whatever")
}
}

Listing B.5: Alignment Actor

class AlignmentActor(replyTo: ActorRef)
  extends Actor with ActorLogging{

  def receive = ready

  def ready: Receive = {
    case Align(query, databaseSequence, score) =>
      Thread.sleep(1)
      val alignmentResult = AlignmentResult(
        Aligner.align(query,
          databaseSequence.sequence,
          score,
          3,
          1,
          true),
        databaseSequence.identifier,
        (query.length *databaseSequence.sequence.length).toLong
      )
      if (alignmentResult == null) {
        throw new RuntimeException()
      }
  }
}
Listing B.6: Query Collector Actor

class QueryCollector(numberOfDatabases: Int,
        batchCollector: ActorRef)
extends Actor with ActorLogging {

  def receive = ready(1, List.empty[QuerySingleDatabaseResult], 0L)

  def ready(receiving: Int,
            results: List[QuerySingleDatabaseResult],
            cells: Long): Receive = {
    case qr: QuerySingleDatabaseResult
        if receiving == numberOfDatabases =>
            log.error("Query complete with {} databases", numberOfDatabases)
            batchCollector ! QueryResult(qr :: results,
                                          cells + qr.cells)
            context.stop(self)
  }
}
Listing B.7: Query Batch Collector Actor

class QueryBatchCollector(numberOfQueries: Int, replyTo: ActorRef, worker: ActorRef) extends Actor with ActorLogging{

def receive =
  ready(1, List.empty[QueryResult])

def ready(receiving: Int, queryResults: List[QueryResult]): Receive = {
  case qr: QueryResult if receiving == numberOfQueries =>
    log.info("Worker finished all queries in batch")
    replyTo ! qr
    replyTo ! QueryBatchResult(qr +: queryResults)
    worker ! WorkComplete
    context.stop(self)
  case qr: QueryResult =>
    log.info("Worker finished query")
    replyTo ! qr
    context.become(ready(receiving+1, qr +: queryResults))

```scala
class QuerySingleDatabaseCollector(replyTo: ActorRef)
  extends Actor with ActorLogging {

  def receive =
    collecting(0,
      List.empty[AlignmentResult],
      0L)

  def collecting(received: Int,
    alignments: List[AlignmentResult],
    cells: Long): Receive = {
    case alignmentResult: AlignmentResult =>
      context.become(collecting(received + 1,
        alignmentResult +: alignments,
        cells + alignmentResult.cells))
    case DatabaseSize(noOfDatabaseSequences) =>
      if received >= noOfDatabaseSequences =>
        log.info("Finished all sequences for database\{\}",
          noOfDatabaseSequences)
        replyTo ! QuerySingleDatabaseResult(alignments, cells)
        context.stop(self)
  case DatabaseSize(noOfDatabaseSequences) =>
    context.become(finishing(noOfDatabaseSequences,
      received + 1,
```
    alignments,
cells))
    case whatever => log.warning("Missed\{\}", whatever)
}

def finishing(expecting: Int,
    received: Int,
    alignments: List[AlignmentResult],
    cells: Long): Receive = {
    case alignmentResult: AlignmentResult
        if received == expecting =>
            log.info("Finished all sequences for database\{\}",
                expecting)
            replyTo ! QuerySingleDatabaseResult(
                alignmentResult +: alignments,
                cells + alignmentResult.cells)
            context.stop(self)
    case alignmentResult: AlignmentResult =>
        context.become(finishing(expecting,
            received +1,
            alignmentResult +: alignments,
            cells + alignmentResult.cells))
    case whatever => log.warning("Missed\{\}", whatever)
}
}
C. R Code Extracts

C.1. R Code Snippets

Listing C.1: Survey data formatting and analysis

```r
library(png)
library(grid)

img <- readPNG("./images/ProcessPyramid.png", TRUE)
grid.raster(img)

library(likert)
library(reshape)

cleanLifeScienceData <- function(results){
  #Remove top line which does not hold data
  results <- results[-1,]
  #Only those who write their own software
  results <- results[results[,57] == "Yes",]
  #Extract the interesting parts of the survey
  results <- results[, c(58:61,93:110)]
  names(results) <- c(1:22)
  # Add a column saying LifeScience
  results["SurveyType"] <- "Life_Scientists"
  return(results)
}
```

324
cleanSoftwareEngineeringData <- function(results){
    # Remove top line which does not hold data
    results <- results[-1,]
    # Extract the interesting parts of the survey
    results <- results[, c(10:13,45:62)]
    # Rename the columns
    names(results)[1] <- "new name"
    names(results) <- c(1:22)
    # Add a column saying SoftwareEngineering
    results["SurveyType"] <- "Software Engineers"
    return(results)
}

likertColumn <- function(data, startColumn){
    levels <- c("Strongly disagree", "Disagree", "Neither agree nor disagree", "Agree", "Strongly agree")
    dataRange <- data[, startColumn]
    # likertTextResults <- do.call(paste, c(dataRange[,], sep=""))
    return(ordered(dataRange, levels))
}

# Extract and clean data
lifeScienceData <-
    read.csv(  
        file="./data/life-science/Results_condensed.csv",  
        head=TRUE, sep="","
    
softwareEngineeringData <-
    read.csv(  
        file="./data/software-engineering/Results_condensed.csv",  
        head=TRUE, sep="","
    
cleanLSData <- cleanLifeScienceData(lifeScienceData)
cleanSEData <-
cleanSoftwareEngineeringData(softwareEngineeringData)
cleanData <- rbind(cleanLSData, cleanSEData)
# Prepare themes for plotting
titleTheme <-
  theme(plot.title = element_text(size=20, face="bold"))
textTheme <- theme(text = element_text(size=18))

# Processes columns
reproducibleBuildColumn <- likertColumn(cleanData, 5)
releaseScriptsColumn <- likertColumn(cleanData, 6)
sourceControlBranchingColumn <- likertColumn(cleanData, 7)
ciWithUnitTestsColumn <- likertColumn(cleanData, 8)
autoSourceAnalysisColumn <- likertColumn(cleanData, 9)
result <- data.frame(reproducibleBuildColumn,
  sourceControlBranchingColumn,
  ciWithUnitTestsColumn,
  releaseScriptsColumn,
  autoSourceAnalysisColumn)
cols <- c("Automated/Reproducible Builds",
  "Source Control Branching",
  "Continuous Integration with Unit Testing",
  "Release Scripts",
  "Automated Source Code Analysis")
colnames(result) <- cols
processes <- likert(result, grouping = cleanData$SurveyType)
title <- "The following processes are used in your organisation ’s projects"
plot(processes, legend = "Profession", centered="true",

326
# Practices columns

unitTestingColumn <- likertColumn(cleanData, 10)
integrationTestingColumn <- likertColumn(cleanData, 11)
uatColumn <- likertColumn(cleanData, 12)
dependencyInjectionColumn <- likertColumn(cleanData, 13)
designPatternsColumn <- likertColumn(cleanData, 14)
codeReviewColumn <- likertColumn(cleanData, 15)
refactoringColumn <- likertColumn(cleanData, 16)
upFrontDesignColumn <- likertColumn(cleanData, 17)
result <- data.frame(unitTestingColumn,
                      integrationTestingColumn,
                      uatColumn,
                      dependencyInjectionColumn,
                      designPatternsColumn,
                      codeReviewColumn,
                      refactoringColumn,
                      upFrontDesignColumn)
cols <- c("Unit\_Testing",
           "Integration\_Testing",
           "User\_Acceptence\_Testing",
           "Dependency\_Injection",
           "Use\_of\_Design\_Patterns",
           "Code\_Review",
           "Refactoring",
           "Up\_Front\_Architecture\_and\_Design")
colnames(result) <- cols
practices <- likert(result, grouping = cleanData$SurveyType)
title <- "The following practices and techniques are used in your organisation's projects"
plot(practices, legend = "Profession",
      centered="true", legend.position = "bottom")
+ ggtitle(title) + titleTheme + textTheme

# Goals columns
scalabilityColumn <- likertColumn(cleanData, 18)
readabilityColumn <- likertColumn(cleanData, 19)
modularityColumn <- likertColumn(cleanData, 20)
performanceColumn <- likertColumn(cleanData, 21)
testabilityColumn <- likertColumn(cleanData, 22)
result <- data.frame(scalabilityColumn, 
                      readabilityColumn, 
                      modularityColumn, 
                      performanceColumn, 
                      testabilityColumn)

cols <- c("Scalability",
          "Readability",
          "Modularity",
          "Performance",
          "Testability")

colnames(result) <- cols
goals <- likert(result, grouping = cleanData$SurveyType)
title <- "The following architecture and design goals are important in your organization"
plot(goals, legend = "Profession",
      centered="true", legend.position = "bottom")
+ ggtitle(title) + titleTheme + textTheme
centered="true", legend.position = "bottom")
+ ggtitle(title)
+ titleTheme
+ textTheme

img <-
  readPNG("./images/BioinformaticProjectRoles.png", TRUE)
grid.raster(img)

img <-
  readPNG("./images/ScientificSoftwareUseScenarios.png", TRUE)
grid.raster(img)