Managing genomic variant calling workflows with Swift/T

Ahmed, Azza E.; Heldenbrand, Jacob; Asmann, Yan; Fadlelmola, Faisal M.; Katz, Daniel S.; Kendig, Katherine; Kendzior, Matthew C.; Li, Tiffany; Ren, Yingxue; Rodriguez, Elliott

Published in:
PLoS ONE

DOI:
10.1371/journal.pone.0211608

IMPORTANT NOTE: You are advised to consult the publisher's version (publisher's PDF) if you wish to cite from it. Please check the document version below.

Document Version
Publisher's PDF, also known as Version of record

Publication date:
2019

Link to publication in University of Groningen/UMCG research database

Citation for published version (APA):
https://doi.org/10.1371/journal.pone.0211608

Copyright
Other than for strictly personal use, it is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license (like Creative Commons).

Take-down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Downloaded from the University of Groningen/UMCG research database (Pure): http://www.rug.nl/research/portal. For technical reasons the number of authors shown on this cover page is limited to 10 maximum.

Download date: 09-11-2019
RESEARCH ARTICLE

Managing genomic variant calling workflows with Swift/T

Azza E. Ahmed1,2☯, Jacob Heldenbrand2☯, Yan Asmann4, Faisal M. Fadelmola1, Daniel S. Katz3, Katherine Kendig3, Matthew C. Kendzior6, Tiffany Li3, Yingxue Ren4, Elliott Rodriguez3, Matthew R. Weber5, Justin M. Wozniak6, Jennie Zermeno3, Liudmila S. Mainzer3,7*.

1 Centre for Bioinformatic s & Systems Biology, Faculty of Science, University of Khartoum, Khartoum, Sudan, 2 Department of Electrical and Electronic Engineering, Faculty of Engineering, University of Khartoum, Khartoum, Sudan, 3 National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign, Urbana-Champaign, Illinois, United States of America, 4 Department of Health Sciences Research, Mayo Clinic, Jacksonville, Florida, United States of America, 5 Department of Crop Sciences, University of Illinois at Urbana-Champaign, Urbana-Champaign, Illinois, United States of America, 6 Argonne National Laboratory, Argonne, Illinois, United States of America, 7 Institute for Genomic Biology, University of Illinois at Urbana-Champaign, Urbana-Champaign, Illinois, United States of America

☯ These authors contributed equally to this work.
* lmainzer@illinois.edu

Abstract

Bioinformatics research is frequently performed using complex workflows with multiple steps, fans, merges, and conditionals. This complexity makes management of the workflow difficult on a computer cluster, especially when running in parallel on large batches of data: hundreds or thousands of samples at a time. Scientific workflow management systems could help with that. Many are now being proposed, but is there yet the "best" workflow management system for bioinformatics? Such a system would need to satisfy numerous, sometimes conflicting requirements: from ease of use, to seamless deployment at peta- and exascale, and portability to the cloud. We evaluated Swift/T as a candidate for such role by implementing a primary genomic variant calling workflow in the Swift/T language, focusing on workflow management, performance and scalability issues that arise from production-grade big data genomic analyses. In the process we introduced novel features into the language, which are now part of its open repository. Additionally, we formalized a set of design criteria for quality, robust, maintainable workflows that must function at-scale in a production setting, such as a large genomic sequencing facility or a major hospital system. The use of Swift/T conveys two key advantages. (1) It operates transparently in multiple cluster scheduling environments (PBS Torque, SLURM, Cray aprun environment, etc.), thus a single workflow is trivially portable across numerous clusters. (2) The leaf functions of Swift/T permit developers to easily swap executables in and out of the workflow, which makes it easy to maintain and to request resources optimal for each stage of the pipeline. While Swift/T’s data-level parallelism eliminates the need to code parallel analysis of multiple samples, it does make debugging more difficult, as is common for implicitly parallel code. Nonetheless, the language gives users a powerful and portable way to scale up analyses in many computing architectures. The code for our implementation of a variant calling workflow using Swift/
T can be found on GitHub at https://github.com/ncsa/Swift-T-Variant-Calling, with full documentation provided at http://swift-t-variant-calling.readthedocs.io/en/latest/.

**Introduction**

Advancements in sequencing technology [1, 2] have paved the way for many applications of Whole Genome Sequencing (WGS) and Whole Exome Sequencing (WES) in genomic research and the clinic [3, 4]. Be it primary variant calling, RNASeq, genome assembly or annotation, a genomics analysis invariably involves constructing a complex workflow that could be hard to manage for large sample sizes (hundreds and beyond, [5–7]) that necessitate the use of large computer clusters. In such cases, features like resiliency and auto-restart in case of node failures, tracking of individual samples, efficient node utilization, and easy debugging of errors and failures are very important. Without a high-quality workflow manager, these requirements can be difficult to satisfy, resulting in error-prone workflow development, maintenance and execution. An additional challenge is porting the workflow among different computing environments, a common need in collaborative and consortium projects.

Monolithic solutions, where a single executable runs the entire analysis, can replace the complex multi-stage workflow and obviate the need for workflow management. Examples of these solutions for primary variant calling include Isaac [8], Genalice [9] and Dragen [10]. These programs offer a plethora of options, but may be too rigid for some analyses, preventing users from swapping algorithms for better accuracy or making adjustments for different species (reference genome, ploidy, known SNP sets etc.) [11]. These monolithic solutions are also developed and maintained by private companies, which may delay or preclude the incorporation of novel approaches and algorithms developed by the scientific and medical community.

Multiple workflow management systems are now available [12] that differ in their design philosophy and implementation. None so far have been found to be the “best” choice for bioinformatics, although some winners are emerging, such as the Common Workflow Language (CWL [13]) and the Workflow Definition Language (WDL [14]), see Discussion. Key distinguishing features are the underlying language and syntax in which the workflow is expressed, and the monitoring and parallel processing capabilities of workflows while executing. Swift/T [15] is one such workflow management system, composed of Swift—a high-level, general-purpose dataflow scripting language [16], and Turbine—a workflow execution engine [17]. The greatest purported advantages of Swift/T are its high portability and ability to scale up to extreme petascale computation levels [18]. Additionally, a number of features make this language an attractive choice for complex bioinformatics workflows [19]:

- *Abstraction and portability*, where cluster resource management is largely hidden from the user, allowing the same code to be seamlessly ported among clusters with different schedulers;

- *Modularity* through the use of leaf functions to define heavyweight processing tasks that are called as need arises;

- *Extensibility* through easy integration of functions written in other languages;

- *Dataflow-based programming framework* that ensures efficient use of compute resources through compile-time optimization for distributed-memory computing models and hybrid parallelism, resulting in high scalability;
• **Code readability** due to its C-like syntax; and

• **Code expressibility**—inclusion of standard programming features, such as conditional execution, iteration, and recursive functions [20].

We explored Swift/T as a choice in the space of currently available workflow management systems. This paper documents our experience implementing, debugging and deploying a genomic variant calling workflow in Swift/T available at https://github.com/ncsa/Swift-T-Variant-Calling and documented on http://swift-t-variant-calling.readthedocs.io/en/latest/.

**Methods and results**

Our chosen use case is genomic variant calling, commonly performed in accordance with the Best Practices established by the GATK team (Genome Analysis Toolkit) [21–23]. It is likely that the GATK will continue to be the standard in research and medicine for those reasons, and also due to the need for HIPAA [24]/CLIA [25] approval and compliance. The GATK is well trusted, validated by the community, and grandfathered in. Thus, the need for a generic, modular and flexible workflow built around the toolkit will persist for some time. We focused only on the primary analysis: the steps from aligning raw reads through variant calling, excluding any downstream steps, such as phasing and annotation. Additionally, we focused on small variant discovery, i.e. the detection of SNPs and InDels, not including structural variant calling. The implementation focused on WGS and WES data. The included functionality was sufficient to test the power and ability of Swift/T and evaluate its usefulness in creating extensible workflows that could be augmented with additional steps.

The variant calling workflow consists of multiple steps that require conditional adjustments based on the analysis use case, such as whole genome vs. exome sequencing, paired- or single-end reads, species or ploidy, etc. The primary role of the workflow management system, such as Swift/T, is to handle this conditional branching and coordinate the launch of command-line tools in accordance with the user-defined configuration and data dependencies, while efficiently managing the computational resources. The underlying workflow language should make it easy to develop and maintain such complex workflows. Based on our prior experience in scaling-up the variant calling workflow [26–28], and that of others [29–31], we have put together a list of requirements to be satisfied while redesigning the workflow in Swift/T, and used them to evaluate the performance of the language for our purposes.

**Workflow design requirements**

**Modularity.** By definition, a workflow is a series of computational tasks, where outputs of one task serve as inputs to the next. Each task can be performed by a selection of bioinformatics software package options driven by the nature of the analysis (Table 1). This flexibility can be enabled by constructing modular workflows, such that each executable is incorporated via a generic wrapper, making it easy for the developer to swap executables at the task level. For example, at the level of the Alignment task, the workflow language should permit easy swapping of BWA MEM [32] for Novoalign [33], conditionally on an option stated in a configuration or run file.

Many tools in Table 1 can take a long time to run on deeply-sequenced samples. This poses a problem for analyses run on computer clusters that have a restrictive maximum job walltime limit. Thus it is useful to break up the workflow into stages—integrated sets of tasks that can be viewed as higher-level modules. Each module is then executed as its own cluster job that fits within the maximum walltime constraint. Chaining such modules together into one executable script effectively requires support for “workflows of workflows”.

---

**Table 1**

<table>
<thead>
<tr>
<th>Tool</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BWA MEM</td>
<td>Sequence alignment tool</td>
</tr>
<tr>
<td>Novoalign</td>
<td>Sequence alignment tool</td>
</tr>
</tbody>
</table>

---

**PLOS ONE** | https://doi.org/10.1371/journal.pone.0211608 | July 9, 2019 | 3 / 20
The modular architecture has additional advantages conferring economy of compute resources and maintainability of code. It allows the user to run a portion of the workflow on the resources optimal for that particular stage, which is useful when a workflow has many fans and merges, but the fans have different node-widths among them. In case of runtime failure, it also enables users to restart the workflow at a failed stage without having to recompute successful upstream calculations. The latter advantage, however, is obviated if the workflow management system itself provides seamless workflow restart from the point of failure—a required feature for complex workflows running at scale. Finally, modularity ensures that the implementation of individual stages can be altered without breaking the workflow, as long as inputs and outputs remain consistent. This way, workflows can be updated with new methodologies as the scientific field and respective tools evolve.

Data parallelism and scalability. A major expectation of a good workflow management system is the ability to develop a single code path that will automatically run in parallel on multiple samples and not force the user to manually code data-level parallelism. This implicit parallelism is not just a matter of convenience, but a significant performance boost. Bioinformatics tools are commonly implemented as multithreaded executables that are not MPI-enabled. Thus, in Bash workflows each task on each sample has to be run as an individual cluster job. If the cluster does not support job arrays, its workload manager can get overwhelmed by the high number of jobs when analyzing large datasets, leading to slow queues or failures. In contrast, a proper workflow management system should run a workflow as a single multi-node job, handle the placement of tasks across the nodes using embedded parallel mechanisms, such as MPI, and scale well with the number of samples.

The workflow manager should also support repetitive fans and merges in the code. For example, in variant calling it is common to cut the walltime of analysis by splitting the input sequencing data into chunks, performing alignment in parallel on all chunks, merging the aligned files per-sample for sorting and deduplication, and finally splitting again for parallel realignment and recalibration per-chromosome (Fig 1, left panel). This pattern of parallelization is more complex than merely running each task on each input sample—yet is a common workflow requirement.

Finally, in bioinformatics we only need certain tools to run on multiple samples in parallel. Other tasks, such as creating folders, user notification or running QC on the whole stage, can and sometimes should be run sequentially. Therefore, it is beneficial to support differential use of data-level parallelism in some modules but not others.
Real-time logging and monitoring. When analyzing many samples at once, especially in a production environment where the data flow continuously through the cluster, it is important to have a good system for logging and monitoring progress of the jobs. At any moment during the run, the analyst should be able to assess (1) which stage of the workflow is running for every sample batch, (2) which samples may have failed and why, (3) which nodes are being used by the analysis, and their health status. Additionally, a well-structured post-analysis record of all events executed on each sample is necessary to ensure reproducibility of the analysis. This can be manually accomplished by developing a system of runtime logs captured via `stdout` dumps, and handling user notification via `mailx`, but both are quite tedious to code for complex, branched, multi-task workflows. A good workflow manager should provide these capabilities implicitly.

Portability. A developer should be able to write a workflow once and then deploy it in many environments: clusters with different node configuration, multiple queues and job schedulers, in HPC or in the cloud. For a workflow as complex as genomic variant calling, having to change and adapt for each different cluster is extremely counterproductive.

Implementation of design requirements in Swift/T

Modularity. The Swift/T language natively supports modularity by defining a “worker” for each executable (“leaf function” in Swift/T terminology), to be called at the appropriate
place in the workflow. For example, we implemented the choice to align reads either using BWA MEM or Novoalign, as follows.

```swift
@dispatch=WORKER
app (file output, file outLog) bwa_mem (string bwaexe, string read1, string read2, string INDEX, string bwamempars[], int PBSCORES, string rgheader)
{
    bwaexe "mem" bwamempars "-t" PBSCORES "-R" rgheader
    INDEX read1 read2 @stdout=output @stderr=outLog;
}
```

```swift
@dispatch=WORKER
app (file output, file outLog) novoalign (string novoalignexe, string read1, string read2, string INDEX, string novoalignparams[], int PBSCORES, string rgheader)
{
    novoalignexe "-c" PBSCORES "-d" INDEX "-f" read1 read2 "-o" "SAM"
    rgheader @stdout=output @stderr=outLog;
}
```

Here each executable is wrapped using the generic "worker" syntax, and workers are conditionally invoked in a compact fashion to perform the Alignment task of the workflow.

```swift
import bioapps.align_dedup;
if (vars["ALIGNERTOOL"] == "BWAMEM")
{
    exec_check(vars["BWAEXE"], "BWAEXE");
    // Directly return the .sam file created from bwa_mem
    outputSam, alignedLog, tmpalignedLog = bwa_mem_logged(vars["BWAEXE"], reads[0], reads[1], vars["BWAINDEX"], [vars["BWAMEMPARAMS"]], threads, rgheader, sampleName);
}
else
{
    // Novoalign is the default aligner
    exec_check(vars["NOVOALIGNEXE"], "NOVOALIGNEXE");
    // Directly return the .sam file created from novoalign
    outputSam, alignedLog, tmpalignedLog = novoalign_logged(vars["NOVOALIGNEXE"], reads[0], reads[1], vars["NOVOALIGNINDEX"], [vars["NOVOALIGNPARAMS"]], threads, rgheader, sampleName);
}
```

Subworkflows, or “stages”, are implemented as individual Swift/T app functions that are chained together by the primary workflow script (Fig 1, right panel). At each stage, the user can direct the workflow to generate the output files necessary for the next stage, or pass on the output generated from a previous run. At the end of each stage, there is an implicit wait instruction that ensures all tasks have finished before the next stage can run (also see next section).

**Data parallelism and scalability.** The “data flow” programming model of Swift/T implicitly supports parallel execution of tasks. Statements are evaluated in parallel unless prohibited by a data dependency or resource constraints, without the developer needing to explicitly code parallelism or synchronization. Swift/T will automatically wait on a process to finish if the next step depends on its output. For example, after read alignment, the step to mark duplicates in an aligned BAM (picard_logged) depends on the previous step (novosort_logged), which produces a sorted BAM (alignedsortedbam) to serve as input to the deduplication step. The essence of implicit parallelization is that picard_logged will wait until novosort_logged is finished due to this data dependency.
Sort
alignedsortedbam, sortLog, tmpnovosortLog = novosort_logged(vars["NOVOSORTEXE"], alignedBam, vars["TMPDIR"], threads, [], string2int(vars["NOVOSORT_MEMLIMIT"]), sampleName);

Mark Duplicates
dedupSortedBam, picardLog, metricsfile, tmppicardLog = picard_logged(vars["JAVAEXE"], vars["JAVA_MAX_HEAP_SIZE"], vars["PICARDJAR"], vars["TMPDIR"], alignedsortedbam, sampleName);

There are some places in the workflow where a stage must wait on another, yet a direct data dependency does not exist. For example, log information begins to be produced right away as the Alignment module begins execution. The output log folder must first exist for this purpose, but the asynchronous parallel execution function of Swift/T may start the Alignment module before it runs the statement to create the log folder. This can be addressed by explicitly forcing the wait either via the “=>” symbol, via wait() statement, or via a dummy variable that “fakes” a data dependency.

mkdir(LogDir) => mkdir(AlignDir) =>
void mkdirSignal = mkdir(tmpLogDir);
wait (mkdirSignal) {
    alignedsam = alignReads(vars, sampleName, reads, rgheader);
}

The above example illustrates the use of a wait() statement, and also the drawbacks of enforcing implicit parallelism across the entire workflow. In bioinformatics, patterns of execution are usually mixed: individual commands running in parallel on many samples are intermixed with serial blocks of code that perform quality control, data management, user notification, or other tasks. It would be useful to have these blocks fenced-off to prevent Swift/T from attempting to run them all asynchronously and in parallel. Parsl, the next step in evolution of Swift language, has that capability [43, 44].

Nonetheless, Swift/T does take care of parallelism in a smart and transparent way that makes efficient use of resources. The user should still take care to request a reasonable number of nodes: too few—and many samples will be processed in series; too many—and resources will be reserved unnecessarily. Beyond that there is no need to worry about task placement, as Turbine will take care of it. This is extremely useful, because bioinformatics programs do not always scale well to the full number of cores available on the compute nodes, and therefore running multiple instances of a task simultaneously on the same node may improve the overall efficiency. For example, BWA MEM normally scales well up to eight threads, so running two eight-thread processes in parallel on a 16-core node is more efficient than running two sixteen-thread processes in series. We implemented this as user-level options that specify the number of cores per node and the number of programs to run on each node simultaneously. From there the workflow determines the number of threads to use for each bioinformatics program, and Swift/T uses Asynchronous Dynamic Load Balancing (ADLB) [45] to distribute those programs across nodes as they become available at run time. Without ADLB one would have to code this explicitly for each job scheduler, which becomes very complicated on clusters that do not support node sharing, i.e. only one job is allowed to run per node. In the latter case a vanilla Bash workflow [46] would need to incorporate an MPI wrapper (e.g. [47]) to take care of program placement across nodes. The MPI backend of ADLB fulfills that function in Swift/T.

We verified correctness of the task dependency chains and parallel execution by tracking start and end times of each task for multiple samples in some of our tests (see next section and Fig 2).

Real-time logging and monitoring. The underlying MPI-based implementation of Swift/T logic makes it possible to leverage standard MPI logging libraries to collect run-time details
about the status of every sample. We used the Message Passing Environment (MPE) library [45] to log the usage of the MPI library itself and ADLB calls [48], and implemented visualization in Jumpshot viewer. To enable such logging requires installation of the MPE library in addition to the standard Swift/T components (C-utils, ADLB library, Turbine and STC). This turned out to be a bit cumbersome because it requires creation of new functions: tcl wrappers around MPE to log when each executable starts and stops.

Another approach to tracking the workflow run time execution is to manually implement Swift/T leaf functions such that the start and end timing of each function are logged. A timing graph can be generated using R script based on this information, showing the analysis steps across samples, chromosomes and specific applications (Fig 2). Interactivity is added via Shiny R package [49]. This is a fairly manual approach, little better than the Bash echo date statements. Nonetheless, it permits one to view the patterns of pipeline execution even if it fails, and partial logs can similarly be viewed as the pipeline is running. To obtain the up-to-date trace, one can type in the R terminal:

```r
if (!require(shiny)) {
  install.packages('shiny')
  library(shiny)
}
runGitHub(repo = "ncsa/Swift-T-Variant-Calling", ref = "master",
  subdir = "src/plotting_app")
```

In conclusion, logging and monitoring can be usefully implemented in a Swift/T workflow, but are not adequately supported at the time of this writing and require quite a bit of work.

**Portability.** Swift/T runs as an MPI program that uses the Turbine [17] and ADLB [45] libraries to manage and distribute the workflow execution on local compute resources (desktop/laptop), parallel computers (clusters/HPCs), and distributed systems (grid/cloud). Its built-in wrappers can launch jobs on many common resource schedulers, such as PBS Torque, Cobalt, Cray aprun, and SLURM [51], using the -m flag passed to the Swift/T executable, i.e. `swift-t -m slurm`. Through these unified wrappers, the user is only left with the trivial task of specifying the required computational resources: queue, memory, wall time, etc.:

```bash
export PPN=<PROGRAMS_PER_NODE>
export NODES=<NUMBER_OF_NODES_TO_RESERVE>
export PROCS=$(($PPN * $NODES))
export WALLTIME=<HH:MM:SS>
```
export QUEUE=<Queue>
export SWIFT_TMP=/path/to/directory/temp
swift-t -m slurm -O3 -n $PROCS
- o /path/to/where/compiled/should/be/saved/compiled.tic
- I /path/to/Swift-T-Variant-Calling/src/
- r /path/to/Swift-T-Variant-Calling/src/bioapps/path/to/Swift-T-Variant-Calling/src/VariantCalling.swift
- runfile=/path/to/your.runfile

We verified both portability and scalability conferred by Swift/T by testing on a variety of HPC systems with a range of cluster setups, job schedulers and patterns of execution (Table 2). Portability across resource schedulers works as expected, although unique setups may require tweaks, such as setting of environmental variables [52], with configuration of 1 sample/node and 2 samples/node.

All other functionality of our workflow was also fully validated on soybean and human Illumina sequencing data, as well as synthetic datasets. The complete list of tested options and features can be found on our GitHub repository [53].

**Robustness against failure.** Swift/T has native support for restarting a task after failure. The user controls the maximum number of allowed retries, and a randomized exponential backoff delay is applied between them, attempting to rerun the task until success or the pipeline terminates, whichever is sooner. Retries do not correct for bugs in the pipeline code, but only for Swift/T leaf function failures that are not related to compilation errors or “assert” failures.

This is useful when applications fail for nondeterministic reasons, such as a filesystem under load slowing down I/O and making the application wait for data, thus causing it to timeout. However, when running wide jobs on large clusters, it is also necessary to have robustness against node failure. In collaboration with the Swift/T team, we introduced the support for moving the retries of the failed task to another, randomly chosen, MPI rank. For reproducibility purposes, random number generation in Swift/T defaults to start from the same seed, which is dependent on the MPI rank where the process is to be evaluated, unless the seed is specified by the turbine variable “TURBINE_SRAND”.

### Table 2. Swift/T delivers on its promise of portability and scalability

<table>
<thead>
<tr>
<th>System</th>
<th>Resource manager</th>
<th>Node type</th>
<th># nodes per run</th>
<th>Node sharing</th>
<th>Test data</th>
</tr>
</thead>
<tbody>
<tr>
<td>iForge [55]</td>
<td>PBS Torque</td>
<td>IvyBridge, 20 cores, 256 GB RAM</td>
<td>1-8</td>
<td>No</td>
<td>Soy NAM [56] using 2, 6, 12, or 16 sample batches †</td>
</tr>
<tr>
<td>XSEDE Stampede2 [57]</td>
<td>Slurm</td>
<td>KNL, 68 cores, 4 hardware threads/core, 96 GB DDR4, 16 GB MCDRAM</td>
<td>1</td>
<td>Yes</td>
<td>GIAB NA12878 Illumina HiSeq Exome (NIST7035) [58]; Synthetic chr1 exome seq 50X</td>
</tr>
<tr>
<td>Bioccluster [50]</td>
<td>Slurm</td>
<td>Dell PowerEdge R620, 24 Cores, 384 GB RAM</td>
<td>1; 3</td>
<td>Yes</td>
<td>Synthetic WES 30X; Synthetic WES 50X; Synthetic WES 70X</td>
</tr>
<tr>
<td>Single server at CBSB, H3ABioNet node</td>
<td>N/A</td>
<td>HP Proliant dl380p gen. 8, 24 cores, 125 G RAM</td>
<td>1</td>
<td>Yes</td>
<td>Synthetic chr1 exome seq 50X</td>
</tr>
</tbody>
</table>

*This Swift/T variant calling workflow was also used on iForge for a variety of analyses on WES and WGS data in different species.*

https://doi.org/10.1371/journal.pone.0211608.t002
Discussion

Complexity of problems in biology means that nearly every kind of analytics is a multi-step process, a pipeline of individual analyses that feed their outputs to each other (e.g. [59–61]). The algorithms and methods used for those processing steps are in continual development by scientists, as computational biology and specifically bioinformatics are still rapidly developing. Few studies can be accomplished via a single integrated executable. Instead, we deal with a heterogeneous medley of software of varied robustness and accuracy, frequently with multiple packages available to perform seemingly the same kind of analysis—yet subtly differing in applicability depending on the species or input data type. Thus bioinformatics today requires advanced, flexible automation via modular data-driven workflows. This is a tall order, considering the added requirements of scalability, portability and robustness. Genomics is a big data field: we no longer talk about sequencing individual organisms, but every baby being born (~ 500 per day per state in the US) and every patient who comes in for a checkup (a million per year in a major hospital), not to mention the massive contemporary crop and livestock genotyping efforts. The workflows managing data analysis at that scale must take full advantage of parallelism on modern hardware, be portable among multiple HPC systems and the cloud, be robust against data corruption and hardware failure, and provide full logging and reporting to the analyst for monitoring and reproducibility.

Recently there has been an incredible upsurge in developing scientific workflow management systems, enough to have resulted in calls for standardization and quality assurance [62]. In this manuscript we reviewed our experience with one such system, Swift/T, touching on workflow management, performance and scalability issues; security was deemed out of scope.

Pros and cons of Swift/T for bioinformatics workflows

Swift/T is a powerful and versatile language that offers many advantages for production large-scale bioinformatics workflows. It allowed us to fulfill most of the requirements outlined in the Requirements section, for variant calling workflow as a use case. Below is our summary of pros and cons based on that experience.

Portability may well be the greatest strength of Swift/T: a workflow written in Swift/T can be executed on a wide variety of compute infrastructures without changing the code, and the user does not need to know about the underlying scheduling environment on the cluster. The language abstracts away the low level concerns such as load balancing, inter-process communication and synchronization of tasks automatically through its compiler (stc) and runtime engine (Turbine), allowing the programmer to focus on the workflow design [63]. Significantly, Swift/T was designed for use in HPC and distributed cluster environments, where the use of containerization is still largely limited due to performance and security concerns. It does not natively support containerization, which would have made porting bioinformatics workflows even more more convenient, as it would have eliminated the need to install all of the (numerous) dependencies. Unfortunately, this is not possible with Swift/T at the moment.

Modularity is another excellent advantage of Swift/T. The language glues together command line tools: either directly by wrapping them in Swift/T app functions if they solely operate on files; or indirectly as tcl packages with corresponding Swift/T app function declarations if they produce numerical or string outputs. Under the hood, Swift/T code is actually compiled into Tcl syntax before Turbine gets to manage the distribution and execution of tasks to compute resources. This further means that wrapping any C, C++ or Fortran application is also easy due to Tcl. This leaf-function modularization, and the ease of integrating code written in other languages into Swift/T environment, is the reason why we preferred this to its predecessor Swift/K [16], which had superior provenance and checkpointing capabilities [64].
Implicit data parallelism and scalability of Swift/T is a powerful way of enabling big data analyses by increasing the amount of simultaneous computation. The language particularly lends itself to use cases that require asynchronous rapid-fire of small, quick parallel jobs [65]. That is one of the many kinds of bioinformatics workloads, but not the most typical one for primary analysis of genomic data. In this field we frequently require a simple wrapper to run a single, time-consuming step on a large number of samples or other units of data level parallelization: i.e. conversion of several thousand BAMs back to FASTQs for reanalysis with the most recent reference genome. However, the data flow task parallelism framework has a substantial learning curve, despite offering familiar control flow statements and expressions in C-like syntax [66]. Coding and debugging can require a more substantial effort than say, Nextflow [67], and that can be a barrier for biologists. An additional inconvenience is that Swift/T does not support piping between applications, which is extensively used in bioinformatics analyses, as they are still overwhelmingly file-based pipelines.

Robustness against failures in Swift/T is supported via leaf function retries, attempting to rerun the task on one of the available ranks. This confers resilience against nondeterministic failures, such as filesystem or cluster interconnect hiccups as well as hardware failures—an important advantage for big data genomics.

Real time logging is provided via runtime Turbine logs, with user-controlled verbosity. These can be quite detailed but challenging to use for debugging when the analyst must understand whether a failure occurred due to data, a bioinformatics application or the Swift/T code bug. The greatest difficulty stems from asynchronous log records, caused by asynchronous execution of tasks. Thus an error printout rarely corresponds to the execution message that immediately precedes it in the log, and finding the failed tasks from the log alone is nearly impossible. We had to manually implement the per-task and per-executable logs in our code, to counteract this inconvenience.

In summary, Swift/T language lends itself to creating highly portable, modular and implicitly parallel workflows. It is very powerful, especially when a workflow consists of raw code pieces written in C, C++, Fortran, etc. However, it may be overkill for those bioinformatics workflows that consist of pre-compiled executables glued together. The lack of support for piping between applications is a major drawback for big-data bioinformatics, resulting in proliferation of intermediary files. Portability, the main advantage of Swift/T, could perhaps be accomplished in simpler ways. In the following sections we review other workflow management systems, to put Swift/T into the broader context of life sciences.

Comparison with GATK reference pipelines

Officially, the GATK provides a set of 2 independent reference pipeline implementations, one for per-sample calling, and the other for joint genotyping across a cohort of samples. These pipelines are written in WDL (https://github.com/gatk-workflows/), which is runnable via Cromwell and Toil (alpha support pre-dates Draft-3 of the language). These reference implementations have been very useful for a large community of bioinformaticians, so we compare them to our Swift/T implementation to highlight the differences among them.

GATK version: Due to early start, our Swift/T pipeline was written with GATK<4 invocations, whereas the GATK reference WDL pipelines leverage GATK4+. However, as discussed above, Swift/T language makes the workflow trivially extensible, such that the switch to GATK4 or addition of further steps can be easily accomplished without the loss of maintainability or ease of deployment.
Analysis stages: user of the Broad’s GATK pipeline can either analyze a single sample from the alignment stage up to producing a gvcf file from the HaplotypeCaller, or jointly analyze the gvcfs of many samples together. The user of our Swift/T pipeline may run complete variant calling for a cohort of samples from alignment to joint calling, or may run a specific stage desired independently by defining the desired analysis stages in a run file. We have not implemented the joint calling, but the repository is open for contributions.

Analysis tools: The reference GATK pipelines assume specific tools for carrying the analysis. The Swift/T implementation was designed to give the user more freedom for specifying tools at each stage. Our implementation makes it easy for the end user to comply with functional equivalence guidelines if desired (or not, depending on the specifics of a given study design).

Language & semantics: The flexibility in choosing tools and analysis stages in our Swift/T pipeline stems from the expressiveness of the Swift/T language itself and the coherence between the language and its execution engine. The two execution engines for running WDL code, Cromwell and Toil, lack support for nested scatter blocks and nested conditionals within scatter blocks, respectively. In other words, parallelization and conditionals are not flexibly supported by Cromwell and Toil.

HPC deployment: For WDL pipelines, Cromwell does work in cluster environments, but has limited scalability in run mode (analysis confined to single node). Supporting the server mode is not attractive to some HPC system administrators for security reasons. In contrast, Swift/T is a language and engine for running analysis on HPC environments, and readily supports a wide range of HPC job schedulers.

Cloud deployment: A motive for WDL and its engines is running analysis jobs in the cloud. In fact, the GATK pipelines implemented by the Broad team are highly cost-optimized for running in both Google Cloud Platform (via FireCloud) and AWS (via AWS batch). On the other hand, Swift/T has less support for usage in cloud environments.

Containerization: Since a main driver for Swift/T development is scalable analysis in HPC environments, it does not readily support containerization technology, nor does our pipeline. WDL on the other hand was developed with an aim to run analysis pipelines in the cloud, and hence containerization is supported via both its engines, Cromwell and Toil.

Challenges in building the “right” workflow manager for computational biology

The implementation of workflow management systems (WMS) for computational biology, bioinformatics and genomics is strongly influenced by culture and prevailing expertise in the multidisciplinary fields. One has to contend with two populations of scientists: those with strong biology background, driven to solve research problems, to whom programming is an unavoidable yet joyless burden; and those able to produce complex and capable code that is not perhaps very user-friendly. This creates a real problem with adoption of any software, including a WMS: the harder it is for a scientist to use a software package compared to an ad-hoc hack, the lower its widespread acceptance in the community [62]. Perhaps that’s why simple glue solutions via Bash, Perl, Python, Make, CMake and similar, have persisted for so long. Their shallow learning curve permits quick production of short-term analytic solutions, which get used over and over despite poor scaling with growing dataset size, and despite requiring a lot of work to port among compute systems.

Scientific Workflow Systems are the next step up from scripting. Those that provide a graphical user interface, such as Taverna [68], Galaxy [69] and Kepler [70] (Table 3), have good accessibility for scientists with less programming experience but require quite a bit of
effort to be set up and maintained, and have limited set of features. In contrast, lower level systems with a command-line interface (CLI), such as Snakemake [71], Luigi [72], BcBio [73], Bpipe [74], are easier to maintain and share, provide good documentation and reproducibility, fault tolerance, and task automation; however, they require a lot more programming expertise.

The cultural gap in capabilities between developers and end users can be closed via implementation of visual programming (GUI-like interface with CLI-like capabilities), thus allowing for customization of analytic tools and technologies with little to no programming background. But, ultimately the right approach to providing scalability and interoperability is probably via implementation of generic low level bioinformatics specific libraries to be used universally across different sets of tools [87].

In the meantime, great strides are being made by the community in trying out different approaches to scientific workflow management and automation, aiming to satisfy the complex requirements [12]:

- seamlessly managing both serial and parallel steps without creating data waits and computational bottlenecks;
- managing complex task dependencies via explicit configuration (e.g. a user-produced XML file in Pegasus [88]), language-specific syntax (BigDataScript [89]), automatic construction of workflow graphs (Swift [16], WDL [14], Nextflow [67]), rule-based approaches (Ruffus [80])

### Table 3. Popular workflow management systems.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Nature</td>
<td>WL¹ and execution engine</td>
<td>WL and execution engine</td>
<td>Web interface</td>
<td>WL and execution engine</td>
</tr>
<tr>
<td>Support community standard WL?</td>
<td>No</td>
<td>No</td>
<td>CWL</td>
<td>No</td>
</tr>
<tr>
<td>User interface</td>
<td>CLI</td>
<td>CLI, REPL [75], IDE [76]</td>
<td>GUI</td>
<td>GUI, CLI, Jupyter notebooks</td>
</tr>
<tr>
<td>Programming paradigm [77]</td>
<td>Dataflow</td>
<td>Dataflow</td>
<td>Sequential [78]</td>
<td>Sequential, dataflow, process network or continuous time [29]</td>
</tr>
<tr>
<td>Containerization support</td>
<td>None</td>
<td>Docker, Singularity</td>
<td>Docker, Singularity</td>
<td>Docker</td>
</tr>
<tr>
<td>Scalability [80]</td>
<td>Extreme scale [81]</td>
<td>Yes</td>
<td>Complicated ‡ [69]</td>
<td>Yes</td>
</tr>
<tr>
<td>Checkpointing and caching</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Portability ‡</td>
<td>Cray aprun, LSF</td>
<td>LSF, NQSI, HTCondor, Kubernetes, Ignite, DNA nexus</td>
<td>LSF, HTCondor, Galaxy Pulsar [82], XSEDE Jetstream [83]</td>
<td>Open stack, Google cloud, Apache Mesos</td>
</tr>
<tr>
<td>Distributed execution</td>
<td>MPI-based</td>
<td>Apache Ignite/ MPI</td>
<td>Spark [84], Hadoop [85]</td>
<td>Spark, Hadoop</td>
</tr>
<tr>
<td>Supported compute architecture</td>
<td>Homogeneous</td>
<td>Homogeneous or heterogeneous</td>
<td>Not clear</td>
<td>Homogeneous or heterogeneous</td>
</tr>
<tr>
<td>Compute resource allocation</td>
<td>Reserved a priori</td>
<td>Reserved a priori</td>
<td>Multiple deployment strategies [86]</td>
<td>Allocated dynamically</td>
</tr>
</tbody>
</table>

¹ WL = workflow language; REPL = Read-Eval-Print-Loop console; CLI = Command Line Interface; GUI = graphical user interface.

‡ Recent optimizations of Galaxy for User interface scalability and Server scalability enable analysis of large datasets for many users.

All these workflow management systems can run on a single server, on clusters managed by PBS, Grid Engine, Slurm, and on AWS.

https://doi.org/10.1371/journal.pone.0211608.t003
and bpipe [74]) or implicit conventions, while abstracting away from HPC cluster management concerns (Job Management System [91]);

- flexibility to work with varied software being run by the workflow (i.e. via containerization), and widely variegated parameter values and configurations (i.e. through workflow autogeneration [92]);

- ability to handle both fixed and user-defined parameters.

The field seems to have converged on a set of relatively widely used workflow languages (WL) to describe the actual flow of computation, and execution engines (EE) that provide automation and portability on HPC environments. Some solutions are by their nature an integrated package of WL+EE (Table 3). However, there has been a widespread recognition of the need to standardize WLs, for the sake of reproducibility—particularly important for clinical applications. Thus separating out an execution engine that could operate on workflows written in a variety of WLs has been very attractive. A few clear leaders have recently emerged: CWL [13] and WDL [14] for workflow definition languages, and Toil [93, 94], Rabix [95] and Cromwell [14] for execution engines (Table 4). CWL enjoys very wide adoption, either being supported, or upcoming support announced among Taverna [68], Galaxy [69], Toil [93], Arvados [96], Rabix [95], Cromwell [14]. To some extent such data-driven workflow languages as CWL and WDL can be viewed as a more advanced step in evolution of a formal scientific workflow. Indeed, when a scientist is only experimenting with the new analysis, it is useful to program it in a powerful lower-level language like Swift, which allows a lot of experimentation with the structure and content of the workflow. Once this has been developed and validated,

Table 4. Popular workflow management systems.

<table>
<thead>
<tr>
<th>Comparison aspect</th>
<th>Toil * [93]</th>
<th>Rabix [95]</th>
<th>Cromwell [14]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nature</td>
<td>Execution engine</td>
<td>Execution engine</td>
<td>Execution engine</td>
</tr>
<tr>
<td>Support community standard WL?</td>
<td>CWL, WDL</td>
<td>CWL</td>
<td>WDL * [97]</td>
</tr>
<tr>
<td>User interface</td>
<td>CLI</td>
<td>GUI *, CLI</td>
<td>CLI</td>
</tr>
<tr>
<td>Containerization support</td>
<td>Docker</td>
<td>Docker</td>
<td>Docker</td>
</tr>
<tr>
<td>Scalability [80]</td>
<td>Petascale</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Checkpointing and caching</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Portability ‡</td>
<td>Open stack, Google Cloud §</td>
<td>LSF, HTCondor, Google JES §</td>
<td></td>
</tr>
<tr>
<td>Distributed execution</td>
<td>Spark</td>
<td>Spark</td>
<td></td>
</tr>
<tr>
<td>Supported compute architecture</td>
<td>Homogeneous or heterogeneous</td>
<td>Homogeneous ‡</td>
<td>Homogeneous ‡</td>
</tr>
<tr>
<td>Compute resource allocation</td>
<td>Allocated dynamically</td>
<td>Reserved a priori §</td>
<td>Reserved a priori</td>
</tr>
</tbody>
</table>

* Toil uniquely has notions of object store and data encryption, which can assure compliance with strict data security requirements.

† Rabix composer (http://docs.rabix.io/rabix-composer-home) is a stand-alone GUI editor for CWL workflows.

‡ In Toil child jobs are executed after their parents have completed (in parallel), and follow-on jobs are run after the successors and their child jobs have finished execution (also in parallel). This creates a Directed Acyclic Graph of jobs to be run, similarly to dataflow. But, unlike in dataflow model, the order of execution depends on whether the parent job has finished and its relation to other jobs, as opposed to whether the data are ready.

§ All these workflow management systems can run on a single server, on clusters managed by PBS, Grid Engine, Slurm, and also on AWS.

Work is ongoing to incorporate support for CWL into Cromwell.

Work is ongoing to also provide support for the GA4GH TES job management system.

https://doi.org/10.1371/journal.pone.0211608.1004
formalizing it in more rigid data-driven framework (CWL, WDL) for reproducibility and later use by non-programmers has a lot of value.

Further efforts toward wider adoption recognize the need to execute biomedical workflows on big data platforms, such as Hadoop and Spark (e.g. Luigi), and the cloud (e.g. Toil, DNA-nexus, SevenBridges, Illumina’s BaseSpace, Curoverse’s Arvados and iPlant Collaborative’s Agave).

Conclusion

Our experience implementing a genomic variant calling workflow in Swift/T suggests that it is a very powerful system for workflow management in supercomputing environments. The language is rich with features that give developers control over their workflow structure and execution while providing familiar syntax. The execution engine also has intelligent mechanisms for task placement and regulation, permitting efficient use of compute resources. This unfortunately comes at the cost of a relatively steep learning curve—a common trade-off for programming languages in general. Thus Swift/T can be an extremely useful—and possibly the best—tool for certain genomics analyses, though its complexity may pose an adoption barrier for biologists.

Acknowledgments

We are grateful for the support of the Blue Waters team, NCSA Industry, and the Argonne/U. Chicago Swift/T developer team during the implementation, testing, and scalability efforts in this project.

This work used Biocluster, the High Performance Computing (HPC) resource for the Carl R Woese Institute for Genomic Biology (IGB) at the University of Illinois at Urbana-Champaign (UIUC). We are grateful for the support by the Computer Network Resource Group (CNRG) while testing the pipeline.

Author Contributions

Conceptualization: Azza E. Ahmed, Jacob Heldenbrand, Liudmila S. Mainzer.

Data curation: Azza E. Ahmed, Jacob Heldenbrand, Yingxue Ren.

Formal analysis: Azza E. Ahmed, Jacob Heldenbrand, Liudmila S. Mainzer.

Funding acquisition: Liudmila S. Mainzer.


Methodology: Liudmila S. Mainzer.

Project administration: Liudmila S. Mainzer.

Resources: Yan Asmann, Justin M. Wozniak, Liudmila S. Mainzer.


Visualization: Azza E. Ahmed, Jacob Heldenbrand.

Writing – original draft: Azza E. Ahmed, Jacob Heldenbrand, Tiffany Li, Jennie Zermen.

Writing – review & editing: Daniel S. Katz, Katherine Kendig, Liudmila S. Mainzer.
References


18. Wozniak JM. Highlights of X-Stack ExM Deliverable Swift/T. Argonne National Lab.(ANL), Argonne, IL (United States); 2016.


55. The University of Illinois at Urbana-Champaign—National Center for Supercomputing Applications. iForge Cluster; 2017. Available from: http://www.ncsa.illinois.edu/industry/iforge.


58. giab_data_indexes: This repository contains data indexes from NIST’s Genome in a Bottle project; 2017. Available from: https://github.com/genome-in-a-bottle/giab_data_indexes.


