NextflowWorkbench: Reproducible and Reusable Workflows for Beginners and Experts.

Jason P. Kurs¹, Manuele Simi¹,², and Fabien Campagne¹,²,³,*

¹The HRH Prince Alwaleed Bin Talal Bin Abdulaziz Alsaud Institute for Computational Biomedicine, Weill Cornell Medicine, New York, NY, United States of America
²Clinical Translational Science Center, Weill Cornell Medicine, New York, NY, United States of America
³Department of Physiology and Biophysics, Weill Cornell Medicine, New York, NY, United States of America
*To whom correspondence should be addressed: fac2003@campagnelab.org

ABSTRACT

Computational workflows and pipelines are often created to automate series of processing steps. For instance, workflows enable one to standardize analysis for large projects or core facilities, but are also useful for individual biologists who need to perform repetitive data processing. Some workflow systems, designed for beginners, offer a graphical user interface and have been very popular with biologists. In practice, these tools are infrequently used by more experienced bioinformaticians, who may require more flexibility or performance than afforded by the user interfaces, and seem to prefer developing workflows with scripting or command line tools. Here, we present a workflow system, the NextflowWorkbench, which was designed for both beginners and experts, and blends the distinction between user interface and scripting language. This system extends and reuses the popular Nextflow workflow description language and shares its advantages. In contrast to Nextflow, NextflowWorkbench offers an integrated development environment that helps complete beginners get started with workflow development. Auto-completion helps beginners who do not know the syntax of the Nextflow language. Reusable processes provide modular workflows. Programmers will benefit from unique interactive features that help users work more productively with docker containers. We illustrate this tool with a workflow to estimate RNA-Seq counts using Kallisto. The workflow can transparently run either on a laptop computer with docker or on a linux cluster. We found that beginners can be taught how to assemble this workflow in a two hours training session. In conclusion, the NextflowWorkbench simplifies the development of reproducible, implicitly parallel workflows. NextflowWorkbench is distributed under the Apache 2.0 license and available at http://workflow.campagnelab.org.

INTRODUCTION

A computational workflow or pipeline is a description of a series of computational steps connected to each other. Each step accepts one or more input(s) and transforms input(s) to produce one or more output(s). Computational workflows are used in many engineering and scientific domains, but are particularly useful in fields such as bioinformatics where analysis activities are repetitive and benefit from being automated. Workflows can be represented as diagrams and their steps followed manually, but many solutions have been developed to represent workflows electronically and automate their execution.

Automated workflow systems include a way for a user to edit the formal representation of the workflow, and its component steps, as well as a runtime system to execute specific workflows. Two broad families of workflow systems have been developed and are still in use today.

The first category is workflows with graphical user interfaces, which often represent workflows as connected components on a 2D diagram. Galaxy and GenePattern are well known examples in Bioinformatics.
We have used the MPS Language Workbench (LWT) in the context of bioinformatics see Simi and Campagne [2014] and Benson and Campagne [2015], implementing the languages using the MPS textgen aspect (Campagne [2014]). Importantly, the typesystem aspect makes it possible to typecheck a workflow as it is being developed and provide feedback to the developer. Nextflow scripts are generated from nodes of Workflow and execution Scripts. Additional developments were conducted by MS and FC to extend these languages and add docker (https://docker.com) and GobyWeb functionality. Full language development logs are available on the GitHub code repository (https://github.com/CampagneLaboratory/NextflowWorkbench) in Campagne [2014]). Briefly, we designed abstractions to represent these scripts to implement different pipelines.

An example of imperative workflow is when bioinformaticians develop a collection of scripts and execute these scripts to implement different pipelines.

While useful, many of these systems fail to fulfill needs that are common in bioinformatics. For instance a makefile cannot easily be run in a distributed manner on multiple nodes of a cluster to parallelize the processing of large collections of data. Scripts can be run in parallel on a cluster with tools like GNU parallel or grid schedulers (e.g., Sun Grid Engine or SLURM), but a recurrent problem with scripts is that most of them are not easily portable to new environments. Indeed, most scripts are written to assume a specific location for the programs that they use (either the script expects a program to be in the PATH, or the script contains a dependency on some location defined in the system where the script was developed). When a dependency is not available in the location assumed by the script, the script fails and the user needs to resolve the issue before retrying execution. The problem is often described as “dependency hell”, a term which many people believe adequately describes the practical difficulties of getting scripts to run on other systems than where they have been developed.

Several groups have recently recognized these problems and have developed improved solutions targeted at bioinformaticians. Recent developments include BigDataScript Cingolani et al. [2015] and Nextflow Di Tommaso et al. [2014] (http://nextflow.io). These solutions address scalability and portability problems and are useful to users with programming and scripting experience.

A declarative language is the makefile input of the make/gmake tool available in UNIX, which was initially developed to automate the compilation of programs, but has been used as well to implement workflows. The second category of workflow systems is based on programming and scripting languages. A workflow is expressed in a declarative or imperative language, or a combination of both. An example of a declarative language is the makefile input of the make/gmake tool available in UNIX, which was initially developed to automate the compilation of programs, but has been used as well to implement workflows. An example of imperative workflow is when bioinformaticians develop a collection of scripts and execute these scripts to implement different pipelines.
Workflow Execution

Workflows can be executed directly from within the MPS LW. Execution is supported on the developers’ machine (for Linux and Mac OS platforms only, since Nextflow does not run on Windows), or on a remote Linux node, where one of the execution mechanisms supported by Nextflow must be available (e.g., Sun Grid Engine, SLURM, Apache Ignite). This capability was implemented with Run Configurations (see Campagne [2015], Chapter 5).

Scripts

To implement scripts as text with auto-completion, we used the MPS RichText plugin (developed and distributed by members of the MBEDDR project Voelter et al. [2012]). The plugin implements the approach described in Voelter [2013].

Documentation

Project documentation has been developed with LaTeX and the Editor2PDF language and plugin (https://github.com/CampagneLaboratory/Editor2PDF). Complete documentation is available at Kurs and Campagne [2015].

RESULTS

A Workflow System for Beginners and Experts

In this study, we designed a workflow system aimed at the full spectrum of workflow users, from beginners to computational experts. Figure 1 presents the advantages of this new workflow system across its intended spectrum of users. The following results section describes the design of this system and the innovations introduced to help with the development and maintenance of reproducible and high-performance workflows. The section also addresses the question of whether this system can be taught effectively to beginners with no command line experience.

Advantages of NextflowWorkbench

<table>
<thead>
<tr>
<th>Beginner</th>
<th>Intermediate</th>
<th>Expert</th>
</tr>
</thead>
<tbody>
<tr>
<td>Training is recommended to facilitate first contact with the platform</td>
<td>Learning is facilitated by the consistency and homogeneity of the platform</td>
<td>Can develop complex Processes and Docker images for others to reuse</td>
</tr>
<tr>
<td>Can use graphical user interface to assemble Workflows. No need to know the Nextflow syntax.</td>
<td>With growing command line experience, develops more complex Processes</td>
<td>Can design language extensions and share with intermediate users and beginners (e.g., declarative GobyWeb resource installation)</td>
</tr>
<tr>
<td>Can use language extensions contributed by experts</td>
<td>As a user learns elements of programming, he/she can extend the languages with intentions to help with repetitive tasks</td>
<td>Can design micro-languages to offer interactive features for more seamless development of Processes and Workflows (e.g., docker path autocompletion)</td>
</tr>
</tbody>
</table>

All the advantages of the Nextflow language: workflows are implicitly parallel & run over several execution systems

All users benefit from a unified platform: graphical notations, integrated version control management, plugin system.

Evaluation of Workflow Systems

We selected Nextflow as the target language for the NextflowWorkbench platform after comparing three systems with similar goals. The comparison included BigDataScript, Nextflow and the Swift language (http://swift-lang.org/). These systems were selected for evaluation because they support parallel execution of workflows on multi-node clusters and provide a reasonable level of abstraction to express workflows.
To compare these systems, two evaluators tried to implement a simple analysis pipeline with each of them. One evaluator was an experienced software engineer with decades of programming and scripting experience (MS). The second evaluator (JPK) was a sophomore undergraduate student with intermediate level programming and scripting experience. Both evaluators completed the implementation of the test pipeline with BigDataScript and Nextflow. The more experienced evaluator developed a partial prototype workflow with Swift, but reported that locating information in the online documentation was tedious and that the semantic of the language was far from intuitive.

This short evaluation revealed a number of characteristics, advantages and drawbacks of the three systems:

Swift requires that all programs be pre-installed on each machine of a cluster. A text file binds the name of his program on the machine. This requirement makes it impossible to develop libraries of reusable processes and the difficulty of knowing at first glance what type of data is exchanged between processes.

The evaluators concluded that of the three systems, Nextflow was the more promising system for representing workflows as scripts.

Requirements for an Improved Workflow Language  Following up this evaluation, we decided to design a variant of the Nextflow language that would directly address the limitations that our evaluation had identified. Specifically, we wanted:

• A modular workflow language that would make it straightforward to reuse processes developed by others in new workflows. Modularity can enable experts to develop processes and share these processes with beginners.

• An explicitly typed workflow language. We believe that an explicitly typed language makes it more obvious to beginners what data are expected as input to a process and what data will be produced as output. Coupled with a mechanism to check type compatibility (a type system) at runtime and highlight type errors, explicit types make it easier for beginners to develop correct workflows. A type system is also useful to experts because it highlights errors that could be missed and only become apparent when trying to execute the workflow.

• A language that does not require the user to know or remember its precise syntax to use it. Such a language could provide auto-completion that guides beginners or even suggests to help beginners in the MetaR project (Campagne et al. [2015]) and were curious to find out if they could also help with workflow development.

Process  A Nextflow Process consists of a set of inputs, a set of outputs, and a workflow script (i.e., outside the script). As standalone language constructs (implemented as MPS root nodes), Processes can be developed and packaged for instance as solutions provided in MPS plugins).

As usual when developing a language with LWT, most parts of a Process can be extended by a user of the language—inside a docker container. This feature is extremely useful to develop workflows that can execute on other systems without tedious dependency installation. Also, one of the limitations of Swift were the lack of language modularity, making it impossible to develop libraries of reusable processes and the difficulty of knowing at first glance what type of data is exchanged between processes.

The next section describes an application of language composition. The next section describes an application of language composition.
Figure 2. NextflowWorkbench Process. A Process defines inputs, outputs, an optional docker container, and a script. In the example shown, the process accepts an input called ‘id’ of type string. The string is used to query the Short Read Archive with the sra-toolkit and retrieve paired FASTQ files. Inputs must be available before the script can execute. Similarly to Nextflow, a process only executes successfully if the outputs it declared have been produced by the script execution. Notice how the Process incorporates graphical elements and colors to clearly mark different roles of the language elements.

When a docker container is specified, as shown in this figure, the commands shown in the script will be executed inside the container. This semantic is implemented by the Nextflow execution runtime. NextflowWorkbench provides autocompletion for input arguments inside the script. This mechanism reduces the risk of typos in variable names and provides instant refactoring of variable names across the script when the workflow programmer renames an input variable.

Processes with Variable Data Resources

Docker containers are useful to isolate the process from the machine where the Process executed, but as we developed bioinformatics workflows with the NextflowWorkbench, we found that docker is only a partial solution when a process requires variable data resources.

As an example, consider the process shown in Figure 3. This Process estimates counts for RNA-seq reads against a transcriptome. Different species have different transcriptomes, and different reference transcriptomes or build versions exist for the same species. As a partial solution, one would create different images for different combinations of species and reference build number. This is not really practical because there are a large number of these combinations and only a few may be of practical interest. If a core facility wanted to provide workflows to support multiple species, core personnel would have to anticipate the needs of its user base and configure a large number of possible images to support any species that the core users could need.

Rather than creating static images with all the data packaged in a container and for all possible choices of interest, it can be more efficient to provide a mechanism to automatically install data resources needed by the script. In this case, assembling the transcriptome resource consists in downloading the appropriate transcriptome reference sequence from Ensembl and indexing this reference with the Kallisto program. We have developed mechanisms to support this on-demand strategy, and make it easy to construct specific data resources. The use of these mechanisms is shown in Figure 3, where a simple requires block declares a dependency on the 

Resource installation scripts are built with the method previously developed for GobyWeb Dorff et al. [2013]. Examples of resources configurations are distributed on GitHub at https://github.com/CampagneLaboratory/gobyweb2-plugins/tree/master/plugins/resources.
Process KallistoCountsWithTuples artifacts/kallisto-homo-sapiens:1.0.0
<<no process option overrides>>

input:
tuple [ file read1, file read2 ]

output:
file 'counts-*.tsv'

script:
requires { resource KALLISTO_INDEX 0.42.3 resolved as: KALLISTO_INDEX -> 0.42.3 }
INDEX,organism = Homo_sapiens
INDEX,reference-build = GRCH38
INDEX,ensembl-version-number = 82
#!/bin/bash (with automatic GobyWeb artifact installation)
echo "Processing: "$read1
TRANSCRIPT_INDEX=${(artifact path KALLISTO_INDEX.INDEX)/transcripts_index}
echo ${TRANSCRIPT_INDEX}
basename=`basename $read1`
echo "Basename=${basename}"

mkdir output
${(artifact path KALLISTO.BINARIES)/bin/kallisto quant --index=${TRANSCRIPT_INDEX} $read1 $read2
--output-dir=./output
#touch output/abundance.tsv
ls -ltrR .
cp output/abundance.tsv counts-${basename}.tsv
exit 0

Figure 3. Process with GobyWeb Artifact Installation. This process uses a special type of script which declares dependencies on GobyWeb resources. GobyWeb resources can automatically install variable data resources, such as a specific transcriptome index identifier by species, reference build and Ensembl version number (as shown). In this example, the script requests installation of the KALLISTO_INDEX resource version 0.42.3. This resource was configured to retrieve the human transcriptome corresponding to GRCH38, in Ensembl version 82. Notice that rather than writing the complicated steps to download and index this transcriptome, the workflow developer can express the data dependency declaratively.
Interactive Docker Features

Developing scripts that run inside a docker container can be challenging because the programmer has to know and remember what programs and data are available inside the container and their precise location. The traditional way to build this understanding is to use interactive console sessions manually started inside a container. The shell can then be used to inspect the files and programs available in the container, and the user has to copy and paste these locations in the script. In NextflowWorkbench, we developed an auto-completion feature that shows container directories interactively when writing the script. With this method, a developer can associate a docker image to a Process, start an interactive container, and use auto-completion to find files or directories of interest (see Figure 4) without ever leaving the workbench. Writing correct paths becomes seamless and no longer requires switching between the editor and console.

Figure 4. Interactive Path Auto-Completion. This figure illustrates interactive auto-completion of paths inside a docker container. After starting an interactive docker container, Process developers can auto-complete paths inside the docker container in the editor. Similar auto-completion functionality is offered for the GobyWeb data and program resources. These interactive features are implemented with standard features of the MPS language workbench.

Figure 5. Diagram of Workflow. This diagram is a schematic representation of the analysis workflow shown in Figure 6. A NextflowWorkbench Workflow consists of a set of inputs, references to processes and a list of optional report clauses. Assume that a user wishes to program a workflow to automate the analysis shown in Figure 5. Figure 6 illustrates how such a workflow can be expressed with the NextflowWorkbench language. To constitute a workflow script, a NextflowWorkbench Workflow accesses to processes by reference. Process references make it possible to name the Process’ inputs and outputs in the context of the workflow. Such names are used to establish connections between process invocations. For instance, in Figure 6, the reference to KallistoCountsWithTuples...
associates the name B to the input of the Process, and associates the name result to its output. When
the name result is defined in this way, the user becomes responsible for the result in the input role
of another process reference.

In order to prevent cyclic dependencies, outputs can be fed to inputs of the same process, but
and can be set on one input at most. When an output is fed to an input of the same process,
duplicate a name (an intention is a context dependent symbol) with the symbols → [ A, B ] indicates
that the output A is associated with the name B, and available through the names A and B. Note that
the brackets to replicate the name as many times as needed.

Download reads from SRA, run FastQC, and estimate counts against the human transcriptome with Kallisto. Produce a
combined counts matrix.

Workflow CombineCounts

with input:
[ "SRR1514132", "SRR1514133", "SRR1514134", "SRR1514135", "SRR1514136", "SRR1514137", "SRR1514138", "SRR1514139",
"SRR1514140", "SRR1514141" ] → [ IDsToDownload, IDsToCombine ]

do:

\[ \text{IDsToDownload} \ll [ \ldots ] \lr \text{id} \rightarrow \text{Download_1M_Reads} \ll \{ [\text{*_1.fastq.gz}, \text{*_2.fastq.gz}] \} \rightarrow ([A, B]) \text{ will run n times} \ll \text{<<options>>}; \]

\[ A \ll [\ldots] \lr \text{[read1, read2]} \rightarrow \text{QC} \lr [\text{*.zip}] \rightarrow \text{zip will run n times} \ll \text{<<options>>}; \]

\[ B \ll [\ldots] \lr \text{[read1, read2]} \rightarrow \text{KallistoCountsWithTuple} \ll \{ \text{counts-*.tsv} \} \rightarrow \text{result} \ll \text{run n times} \ll \text{<<options>>}; \]

\[ \text{result.toList()} \rightarrow \text{tsv} \rightarrow \text{CombineCounts} \rightarrow \text{counts.tsv} \rightarrow \text{combined} \ll \text{will run 1 time} \ll \text{<<options>>}; \]

\[ \text{IDsToCombine.toList()} \rightarrow \text{ids} \]

and report:
\[ [\ldots] \]

**Figure 6. Workflow Example.** In this example, a set of SRA identifiers is defined in the input section of
the workflow. The list is then duplicated and fed to two processes: Download_1M_Reads and
CombineCounts. The Download_1M_Reads will run once for each identifier. When this process
terminates, the output, which consists of the tuple of files ["*_1.fastq.gz", "*_2.fastq.gz"], is duplicated to
be fed to the QC and KallistoCountsWithTuple processes. The counts obtained with Kallisto are
fed to CombineCounts along with the list of ids to produce a combined matrix of counts for all the
samples analyzed.

Utility to Non-Programmers

An important question is whether the user interface provided by Nextflow-workbench is sufficient to help non programmers with a biology or clinical background
develop workflows. Workbench provides sufficient assistance to help non programmers with a biology or clinical background
install the software on the trainees machines. About 30-45 minutes of the training sessions are spent verifying
installation steps and provisioning a cluster for remote execution of workflows would go a long way to
make this training more accessible and we are actively developing solutions to this end.

In order to prevent cyclic dependencies, output names cannot be set on an input of the same process,
and can be set on one input at most. When an output is fed to an input of the same process,
duplicate a name (an intention is a context dependent symbol) with the symbols → [ A, B ] indicates
that the output A is associated with the name B, and available through the names A and B. Note that
the brackets to replicate the name as many times as needed.

Download reads from SRA, run FastQC, and estimate counts against the human transcriptome with Kallisto. Produce a
combined counts matrix.

Workflow CombineCounts

with input:
[ "SRR1514132", "SRR1514133", "SRR1514134", "SRR1514135", "SRR1514136", "SRR1514137", "SRR1514138", "SRR1514139",
"SRR1514140", "SRR1514141" ] → [ IDsToDownload, IDsToCombine ]

do:

\[ \text{IDsToDownload} \ll [ \ldots ] \lr \text{id} \rightarrow \text{Download_1M_Reads} \ll \{ [\text{*_1.fastq.gz}, \text{*_2.fastq.gz}] \} \rightarrow ([A, B]) \text{ will run n times} \ll \text{<<options>>}; \]

\[ A \ll [\ldots] \lr \text{[read1, read2]} \rightarrow \text{QC} \lr [\text{*.zip}] \rightarrow \text{zip will run n times} \ll \text{<<options>>}; \]

\[ B \ll [\ldots] \lr \text{[read1, read2]} \rightarrow \text{KallistoCountsWithTuple} \ll \{ \text{counts-*.tsv} \} \rightarrow \text{result} \ll \text{run n times} \ll \text{<<options>>}; \]

\[ \text{result.toList()} \rightarrow \text{tsv} \rightarrow \text{CombineCounts} \rightarrow \text{counts.tsv} \rightarrow \text{combined} \ll \text{will run 1 time} \ll \text{<<options>>}; \]

\[ \text{IDsToCombine.toList()} \rightarrow \text{ids} \]

and report:
\[ [\ldots] \]

**Figure 6. Workflow Example.** In this example, a set of SRA identifiers is defined in the input section of
the workflow. The list is then duplicated and fed to two processes: Download_1M_Reads and
CombineCounts. The Download_1M_Reads will run once for each identifier. When this process
terminates, the output, which consists of the tuple of files ["*_1.fastq.gz", "*_2.fastq.gz"], is duplicated to
be fed to the QC and KallistoCountsWithTuple processes. The counts obtained with Kallisto are
fed to CombineCounts along with the list of ids to produce a combined matrix of counts for all the
samples analyzed.

Utility to Non-Programmers

An important question is whether the user interface provided by Nextflow-workbench is sufficient to help non programmers with a biology or clinical background
develop workflows. Workbench provides sufficient assistance to help non programmers with a biology or clinical background
install the software on the trainees machines. About 30-45 minutes of the training sessions are spent verifying
installation steps and provisioning a cluster for remote execution of workflows would go a long way to
make this training more accessible and we are actively developing solutions to this end.

In order to prevent cyclic dependencies, output names cannot be set on an input of the same process,
and can be set on one input at most. When an output is fed to an input of the same process,
duplicate a name (an intention is a context dependent symbol) with the symbols → [ A, B ] indicates
that the output A is associated with the name B, and available through the names A and B. Note that
the brackets to replicate the name as many times as needed.

Download reads from SRA, run FastQC, and estimate counts against the human transcriptome with Kallisto. Produce a
combined counts matrix.

Workflow CombineCounts

with input:
[ "SRR1514132", "SRR1514133", "SRR1514134", "SRR1514135", "SRR1514136", "SRR1514137", "SRR1514138", "SRR1514139",
"SRR1514140", "SRR1514141" ] → [ IDsToDownload, IDsToCombine ]

do:

\[ \text{IDsToDownload} \ll [ \ldots ] \lr \text{id} \rightarrow \text{Download_1M_Reads} \ll \{ [\text{*_1.fastq.gz}, \text{*_2.fastq.gz}] \} \rightarrow ([A, B]) \text{ will run n times} \ll \text{<<options>>}; \]

\[ A \ll [\ldots] \lr \text{[read1, read2]} \rightarrow \text{QC} \lr [\text{*.zip}] \rightarrow \text{zip will run n times} \ll \text{<<options>>}; \]

\[ B \ll [\ldots] \lr \text{[read1, read2]} \rightarrow \text{KallistoCountsWithTuple} \ll \{ \text{counts-*.tsv} \} \rightarrow \text{result} \ll \text{run n times} \ll \text{<<options>>}; \]

\[ \text{result.toList()} \rightarrow \text{tsv} \rightarrow \text{CombineCounts} \rightarrow \text{counts.tsv} \rightarrow \text{combined} \ll \text{will run 1 time} \ll \text{<<options>>}; \]

\[ \text{IDsToCombine.toList()} \rightarrow \text{ids} \]

and report:
\[ [\ldots] \]
Build Image images: Kallisto_Image
Dockerfile Kallisto {
FROM artifacts / software- gcc4.8 : 1.3.0
MAINTAINER Campagnelab *manuele.simi@campagnelab.org *
install gobyweb artifacts { resource KALLISTO_INDEX 0.42.3 resolved as: KALLISTO_INDEX -> 0.42.3 }
INDEX . organism = Homo_sapiens
INDEX . reference-build = GRCH38
INDEX . ensembl-version-number = 82
}
}

Figure 7. Support for Building Docker Images. NextflowWorkbench offers a composable language to help workflow programmers construct Docker images. In this example, we show a DockerFile root node with a special instruction type called install gobyweb artifacts. This special instruction generates RUN commands that install GobyWeb resources in the docker image. Pressing the Build Image button assembles the image. Built images can be used in Processes. Such instructions are useful to create frozen docker images that contain a specific data resource, for instance for clinical analysis workflows which must be frozen as required by regulations.

DISCUSSION

Tools to support the development and execution of workflows have been popular among scientists who need to automate repetitive execution of programs to process different inputs. Galaxy and GenePattern are examples of such tools, which we call graphical workflow systems, designed to help biologists who are not programmers take advantage of bioinformatics programs and automate analyses. The tools offer a graphical metaphor for a workflow where boxes represent tools and lines connect the tools where data are fed from one tool to the other. Graphical workflow systems in wide use today were introduced about 10 years ago Giardine et al. [2005], Reich et al. [2006], Hull et al. [2006]. While popularity for these tools grew among computational beginners, experts have yet to widely adopt these systems. While it is unclear what set of reasons explain this lack of interest across the community, understanding these reasons could be useful to developers of graphical workflow systems.

Advanced Docker Features

The NextflowWorkbench can be used as an interactive development environment to help develop docker images. We have developed a composability language that allows docker build files to be written in the workbench to assemble a docker image. In this example, we show a DockerFile root node with a special instruction type called install gobyweb artifacts. This special instruction generates RUN commands that install GobyWeb resources in the docker image. Pressing the Build Image button assembles the image. Built images can be used in Processes. Such instructions are useful to create frozen docker images that contain a specific data resource, for instance for clinical analysis workflows which must be frozen as required by regulations.

Kronos is a recent workflow system presented in a pre-print Taghiyar et al. [2016]. Kronos supports writing workflow as structured text files that get compiled into python scripts for execution on a variety of computational platforms (local execution, cluster and cloud). Kronos also offers a graphical interface to define tasks (analogous to Nextflow processes) and I/O connections. The Nextflow language supports explicit parallelization. The focus of these systems is on helping expert bioinformaticians build high performance parallel data analysis workflows. In addition to the workflow language, these systems offer a runtime system to execute the workflow using a range of high-performance grid schedulers (such as Sun Grid Engine, SLURM or Apache Ignite). Because these systems require expressing workflows in text source code for a custom language, learning how to use them requires becoming familiar with the syntax of a new computational language. This is one reason why Kronos and other systems with prior scripting experience. In contrast to these systems, NextflowWorkbench can be thought of as an integrated development environment that provides the user with full language support with auto-completion to guide new users while they learn the language. Kronos is a recent workflow system presented in a pre-print Taghiyar et al. [2016]. Kronos supports writing workflow as structured text files that get compiled into python scripts for execution on a variety of computational platforms (local execution, cluster and cloud). Kronos also offers a graphical interface to define tasks (analogous to Nextflow processes) and I/O connections. The Nextflow language supports explicit parallelization. The focus of these systems is on helping expert bioinformaticians build high performance parallel data analysis workflows. In addition to the workflow language, these systems offer a runtime system to execute the workflow using a range of high-performance grid schedulers (such as Sun Grid Engine, SLURM or Apache Ignite). Because these systems require expressing workflows in text source code for a custom language, learning how to use them requires becoming familiar with the syntax of a new computational language. This is one reason why Kronos and other systems with prior scripting experience. In contrast to these systems, NextflowWorkbench can be thought of as an integrated development environment that provides the user with full language support with auto-completion to guide new users while they learn the language.
simple I/O connections because it is possible to transform data produced by processes with a sequence of functions (e.g., see the use of the `toList()` function in Figure 6) before the data is provided to a process. A large number of pre-defined functions are supported by Nextflow as well as user-defined closures that can be used to process data in custom ways. This capability is not clearly apparent in the version of the Kronos preprint available as of this writing. Kronos also aims to provide a system that both beginners and experts can use, but does not provide a well-integrated development environment. Despite its simplicity, the Kronos language represent a new text-based language whose syntax must be learned by beginners who will find out about errors when running the compiler. In contrast, NextflowWorkbench provides an interactive graphical user interface and advanced interactive features, such as real time error detection, on top of an expressive workflow language: Nextflow.

In a broader context, languages to express workflows are essential elements of infrastructures where data analyses are moved to the data. Such infrastructures are becoming necessary in situations where volumes of data are very large (i.e., a tera-byte and more). When several groups need to make computations on the same set of data, transferring the data to the analysis code becomes a bottleneck. In these cases, it is more efficient to move the code to the data than to do the opposite. In the USA, the National Cancer Institute at the National Institutes of Health has started a series of pilots to evaluate this type of infrastructure. The Broad institute, who leads one of these pilot infrastructure projects, has developed the Workflow Description Language and supports executing workflows expressed in WDL on the Broad infrastructure. Another, similar, but incompatible file format to express workflows is the Common Workflow Language (CWL), developed by the Seven Bridges Cancer Genomics Cloud in another NIH/NCI cloud pilot. CWL aims to become a widely used standard to express workflows. To this end, the project organizers are trying to engage a large community of people in the design of CWL. NextflowWorkbench differs from these efforts in several important ways. First, the focus is on user experience to enable beginners to develop and use workflows and to make experts more productive. Neither WDL or CWL address this need. Second, because we used LWT to develop NextflowWorkbench, others can develop extensions of the languages that become immediately integrated with NextflowWorkbench (through language composition and micro-language design, as illustrated in Campagne and Simi [2015]). In our experience, in most cases, there is no need for coordination with our group to develop simple extensions and share them with others. This differs strikingly from a standard development effort, which requires numerous formal communications and coordination before any change can be made to the specification of the “standard”.

**CONCLUSION**

In this study, we presented the design and implementation of a workflow system meant for a broad spectrum of potential users, ranging from computational beginners to expert bioinformaticians. We applied language workbench technology to develop such a system, using Nextflow as underlying workflow execution system. We found that we could successfully teach this new workflow system to non-programmers who are able to develop and reuse the simple workflow presented in this manuscript in a short training session of two hours.

**ACKNOWLEDGMENTS**

The authors thank Paolo Di Tommaso for assistance integrating changes needed for the Nextflow-Workbench into Nextflow distributions. We thank the members of Cédric Notredame’s Laboratory (http://www.crg.eu/cedric_notredame) for developing and maintaining Nextflow as an open-source project. This investigation was supported by the National Institutes of Health NIAID award 5R01AI107762-02 to Fabien Campagne and by grant UL1 RR024996 (National Institutes of Health [NIH]/National Center for Research Resources) of the Clinical and Translation Science Center at Weill Cornell Medical College. We thank the training session participants who have provided feedback on earlier versions of NextflowWorkbench and help make this new workflow system relevant to computational beginners.

**REFERENCES**


