Imperial College London

MSc Bioinformatics and Theoretical Systems Biology

Computing Project Group Report

RAWG: RNA-Seq Analysis Workflow Generator

Authors
Alessandro Pio Greco (APG)
Patrick Hedley-Miller (PHM)
Filipe Jesus (FJ)
Zeyu Yang (ZY)

Supervised by
Dr Derek Huntley

May 8, 2019

Abstract

Written by APG and ZY

Motivation: RNA sequencing (RNA-Seq) is becoming the golden standard for analysing gene expressions in biological samples. The development of sequencing technologies has greatly improved the productivity of gathering data from cells over older technologies, such as micro-arrays. With the ever increasing sequencing speed and reduced cost, the amount of data produced is growing exponentially which imposes new challenges on bioinformatics data analysis. Many analysis pipelines were developed, enabling standardisation and automation of RNA-Seq data analysis. However, most of the pipelines relies on command-line interface which can be difficult for end-user to learn and use.

Result: We present a complete RNA-Seq data analysis framework with emphases on care-free and ease of use. Our framework uses contemporary workflow description standard, Common Workflow Language, as the analysis pipeline foundation. A website, based on the Django framework, is designed for user to upload data and submit analyses with a few clicks. The end result is a flexible data analysis platform that is not only easy to use but also improves data reproducibility and provenance in biological science. The code from this project is freely available at http://www.github.com/rawgene under Apache License 2.0.

Acknowledgement

We would like to thank Dr Derek Huntley for his guidance and for giving us the freedom in deciding this project's direction. We are thankful for the neuroblastoma data generated by Dr Ximena Montano. Gratitude is also due to Dr Suhail Islam for his technical support and patience.

Contents

Abbreviation

cDNA	Complementary DNA
CSV	Comma Seperated Values
CWL	Common Workflow Language
bp	base pair
DGE	Differential Gene Expression
DEE	Differential Exon Expression
DIE	Differential Isoform Expression
DNA	Deoxyribonucleic Acid
FDR	False Discovery Rate
fGSEA	fast Gene Set Enrichment Analysis
FM-index	Full-text index in Minute space
FPKM	Fragments Per Kilobase of transcript per Million mapped reads
GATK	Genome Analysis Toolkit
GO	Gene Ontology
GSEA	Gene Set Enrichment Analysis
HTSeq	High-Throughput Sequencing
HPC	High Performance Computer
JSON	JavaScript Object Notation
miRNA	micro RNA
MISO	Mixture of Isoforms
mRNA	messenger RNA
ORM	Object-Relational Mapper
ORCID	Open Researcher and Contributor ID
PC	Principle Component
PCA	Principle Component Analysis
RNA	Ribonucleic Acid
RNA-Seq	RNA Sequencing
RAWG	RNA-Seq Analysis Workflow Generator
siRNA	small interfering RNA
SMRT	Single Molecule Real-Time Sequencing
STAR	Spliced Transcripts Alignment to a Reference
SVG	Scalable Vector Graphics
SQL	Structured Query Language
UUID	Universally Unique Identifier

1 Introduction

1.1 RNA-Seq *Written by FJ*

Since the dawn of high-throughput sequencing, trends have moved on from whole genome sequencing to whole transcriptome sequencing. The popularity of transcriptomics can be explained by both the messenger like and regulatory role of RNA itself[1]. Protein is the product of translation, therefore the quantification of mRNA can give an idea of the proteome of a cell, which is difficult to measure directly[2]. In the same vein, DNA is a precursor to RNA, thus the structural variants of RNA can give insight to possible mutations in the DNA code and overall errors in the gene expression pathway of the cell. Also, siRNA, miRNA and other forms have been known to be key regulatory units to gene expression and, consequently, the quantification of these molecules gives further information into the current life cycle of the cell. The above reasons showcase the power of transcriptomics and explain its popularity in both academia and industry.

Improvements in technology have aided the transcriptomic revolution. Next-generation sequencing technologies such as Roche 454[3], Solexa[4], Nanopore[5] and Single Molecule Real-Time Sequencing (SMRT)[6] have made whole transcriptome sequencing fast, cheap and easy to use, and have pioneered the age of RNA-Seq analysis. Their predecessor, the micro-array, is still popular in some labs but lacks the sequence information which allows powerful analysis with RNA-Seq[7]. Unlike the sequencing itself, the state of current tools for RNA-Seq analysis can be described as complex to understand and run, with multiple alternative tools for each stage of the process being available along with many data standards. Thus there is no single best workflow for RNA-seq data analysis.

Our aim in this project was to develop a framework, RNA-Seq Analysis Workflow Generator (RAWG), which can act as a one stop shop for anyone wanting to preform RNA-Seq differential expression analysis. We have broken down a basic RNA-Seq analysis into four key steps; alignment, assembly, differential expression analysis and post-analysis. Descriptions of each step are summarised in Table 1. Our framework consists of a user interface and server-side scripts for conducting the analysis. The user will create a workflow by selecting from a range of gold-standard tools for each step in the front-end interface. The webportal will also provide a variety of analysis options including differential gene, transcript, exon and isoform expression.

Step	**Definition**
Alignment	Align sequences in FASTQ files to assembled genomes/transcriptomes. Each tool will use unique strategies in order to improve accuracy and/or speed.
Assembly	Assemble alignments into non-overlapping transcripts. Quantifies the per transcript reads.
Differential Expression Analysis	Statistical tools for the calculation of expression fold change between sample groups of interest and testing the significance of change.
Post-analysis	Visualisation of the differential expression results and normalised counts. Also includes gene set enrichment analysis which determines whether a set of genes shows statistical significance, concordant differences between two groups of interest.

Table 1: Explanation of each of the four basic steps in a RNA-Seq analysis.

1.2 Common Workflow Language *Written by ZY*

Our design goal is to have a flexible framework that can generate RNA-Seq analysis workflows based on the user's selection of command-line tools. The framework should be extensible so researchers can integrate other tools based on needs. In addition, we wanted to explore the problem of data reproducibility in computational science and use the framework to compare between different data pipelines. Many workflow management systems are used in the bioinformatics community, however they commonly lack interoperability, portability or provenance[8].

Common Workflow Language (CWL) is developed as an open standard to serve data intensive science and is increasingly adopted by bioinformaticans[9]. CWL leverages container technology (support Docker, Singulaity, uDocker, etc.) to ease the setup of local software environments and dependencies, and to achieve workflow portability and data reproducibility. It is important to note that CWL is a specification to describe the connections between tools and many software (runners) can be used to run the workflow. Depending on the workflow runner, CWL workflows can be executed in a variety of computational environments, ranging from workstations and single-node servers to clusters, high performance computers (HPC), and private or commercial cloud (such as Amazon Web Services, Microsoft Azure and Google Compute Engine). A general depiction of CWL workflow platform architecture is shown in Figure 1.

CWL documents consist of two parts, the workflow description file and the input file. This two-part system improves workflow reusablility and portability as researchers can conveniently share their workflow via a single file. Indeed, many research groups publish their data pipelines on code hosting sites such as GitHub, Bitbucket and GitLab. Dockstore, a dedicated registry for sharing

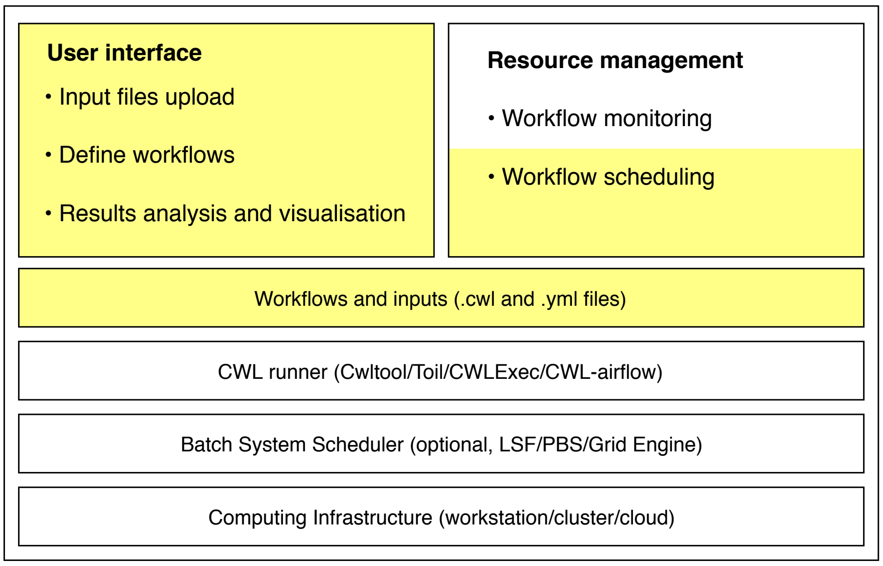

User interface
• Input files upload
• Define workflows
• Results analysis and visualisation
Resource management
• Workflow monitoring
• Workflow scheduling
Workflows and inputs (.cwl and .yml files)
CWL runner (Cwltool/Toil/CWLExec/CWL-airflow)
Batch System Scheduler (optional, LSF/PBS/Grid Engine)
Computing Infrastructure (workstation/cluster/cloud)

Figure 1: Illustration of a typical CWL workflow platform architecture. This project's premise is highlighted in yellow.

computational research software, stores hundreds of containerised tools with workflows described in CWL and other formats[10].

Data provenance, defined as the computationally generated data's derivative history, is gaining more attention each year as more computational research data is put under scrutiny[11]. Utilising CWLProv, CWL brings the possibility of recording the complete provenance metadata of workflows as a Research Object[12]. CWL provides options to record the Docker container's identifier as well as the user's ORCID. Since CWL's inception in 2014, many pre-existing analysis systems have adopted the CWL standard and provided support for running CWL workflows either natively or by converting CWL files to another format. Major analysis systems include Apache Airflow, Toil, Rabix Bundy with more currently implementing support[13, 14, 15].

CWL provides many benefits over other workflow description languages including portability, reproducibility, reusability and data provenance. CWL workflow description file's modular structure is essential for generating workflows programmatically. Hence, CWL is our ideal choice for the framework's underlying workflow system.

1.3 Hosted Bioinformatics Workflow System *Written by FJ*

Web based bioinformatics workflow systems are already available and they all occupy their own niches with dedicated user bases. The common aim between these systems is to provide a user friendly way to run bioinformatics software, however they all have their own approaches.

Galaxy is one of the most popular scientific workflow tools [16]. As a platform, Galaxy solves the problem of workflow generation by providing a GUI and the ability for users to create their own "recipes" (workflows). These recipes can then be published for other users to use. Galaxy's popularity stems from its accessibility, reproducibility and transparency. Its openness and flexibility in recipes means it is a host for many more areas of analysis aside from RNA-Seq. However, due to the nature of Galaxy, the user must create their own workflow if attempting a novel combination of tools. This requires the user to parameterise each tool and provide the links between tools in the workflow using graph based visual programming (Figure 2). This process can be as complex as running tools on the command line and thus can be cumbersome for first time users.

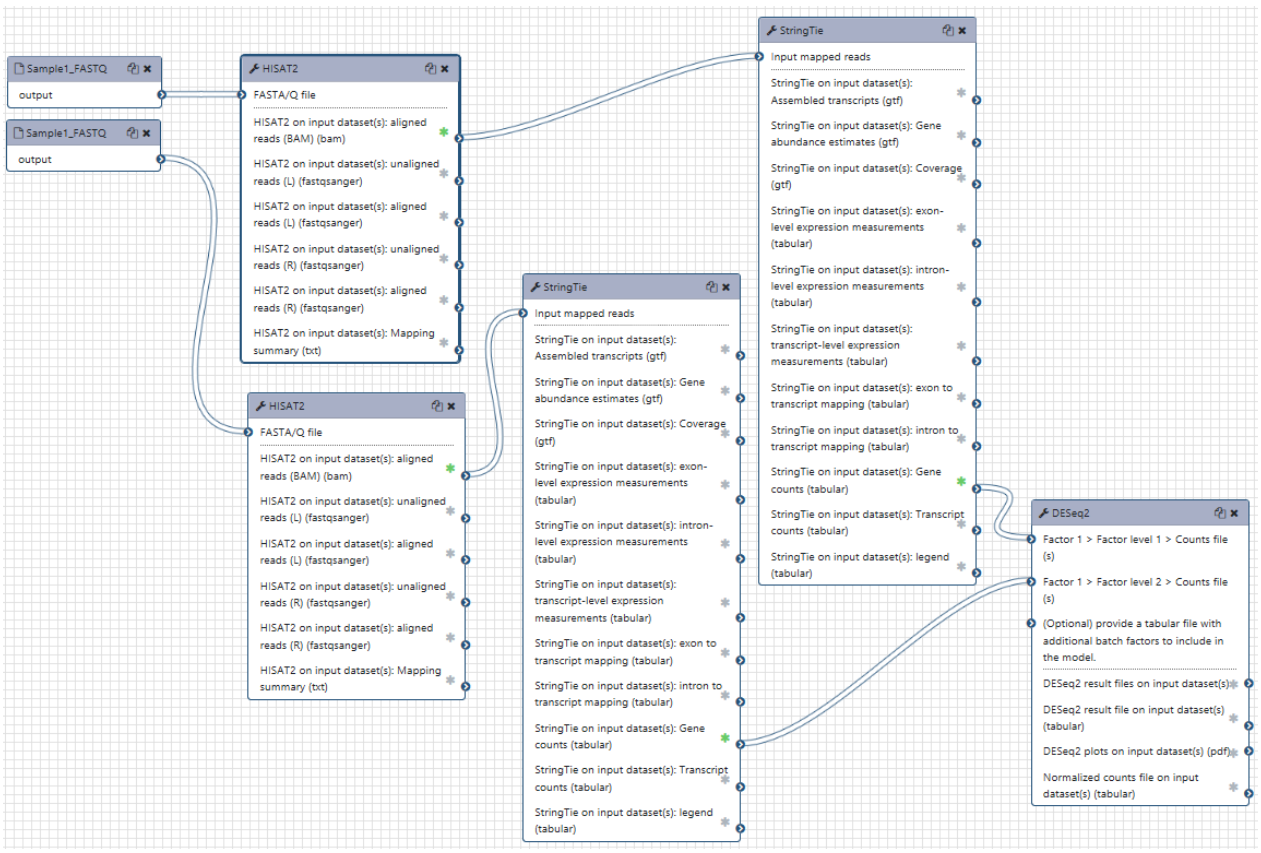

Sample1_FASTQ
output
Sample1_FASTQ
output
HISAT2
FASTA/Q file
HISAT2 on input dataset(s): aligned reads (BAM) (bam)
HISAT2 on input dataset(s): unaligned reads (L) (fastqsanger)
HISAT2 on input dataset(s): aligned reads (L) (fastqsanger)
HISAT2 on input dataset(s): unaligned reads (R) (fastqsanger)
HISAT2 on input dataset(s): aligned reads (R) (fastqsanger)
HISAT2 on input dataset(s): Mapping summary (txt)
HISAT2
FASTA/Q file
HISAT2 on input dataset(s): aligned reads (BAM) (bam)
HISAT2 on input dataset(s): unaligned reads (L) (fastqsanger)
HISAT2 on input dataset(s): aligned reads (L) (fastqsanger)
HISAT2 on input dataset(s): unaligned reads (R) (fastqsanger)
HISAT2 on input dataset(s): aligned reads (R) (fastqsanger)
HISAT2 on input dataset(s): Mapping summary (txt)
StringTie
Input mapped reads
StringTie on input dataset(s): Assembled transcripts (gtf)
StringTie on input dataset(s): Gene abundance estimates (gtf)
StringTie on input dataset(s): Coverage (gtf)
StringTie on input dataset(s): exon-level expression measurements (tabular)
StringTie on input dataset(s): intron-level expression measurements (tabular)
StringTie on input dataset(s): transcript-level expression measurements (tabular)
StringTie on input dataset(s): exon to transcript mapping (tabular)
StringTie on input dataset(s): intron to transcript mapping (tabular)
StringTie on input dataset(s): Gene counts (tabular)
StringTie on input dataset(s): Transcript counts (tabular)
StringTie on input dataset(s): legend (tabular)
StringTie
Input mapped reads
StringTie on input dataset(s): Assembled transcripts (gtf)
StringTie on input dataset(s): Gene abundance estimates (gtf)
StringTie on input dataset(s): Coverage (gtf)
StringTie on input dataset(s): exon-level expression measurements (tabular)
StringTie on input dataset(s): intron-level expression measurements (tabular)
StringTie on input dataset(s): transcript-level expression measurements (tabular)
StringTie on input dataset(s): exon to transcript mapping (tabular)
StringTie on input dataset(s): intron to transcript mapping (tabular)
StringTie on input dataset(s): Gene counts (tabular)
StringTie on input dataset(s): Transcript counts (tabular)
StringTie on input dataset(s): legend (tabular)
DESeq2
Factor 1 > Factor level 1 > Counts file (s)
Factor 1 > Factor level 2 > Counts file (s)
(Optional) provide a tabular file with additional batch factors to include in the model.
DESeq2 result files on input dataset(s)
DESeq2 result file on input dataset(s) (tabular)
DESeq2 plots on input dataset(s) (pdf)
Normalized counts file on input dataset(s) (tabular)

Figure 2: Galaxy workflow creation. Example of graph based visual programming. A workflow can be considered a directed acyclic graph as the user connects the outputs of one tool (node) to the inputs of another.

Genevestigator is a commercial application for RNA-Seq database curation and analysis[17]. Its niche is manual database curation in which workflows are run and gene ids/conditions are standardised allowing for powerful between-study comparisons. However, due to its closed and manual nature, the user will be limited in workflow choices and run-time. The manual analysis of users' data by Genevestigator employees also calls data security and provenience into question, however they do guarantee privacy and no issues have ever been reported.

SevenBridges is a commercial workbench platform similar to Galaxy with better performance results[18]. They focus on generating workflows for bioinformatics analysis. Specifically they have a wide range of tools for RNA-Seq analysis allowing for a broad range of workflows, including fusion gene and single cell. SevenBridges also utilises CWL to describe their workflows and allows the user to use their own CWL scripts. However, similar to Galaxy, the workflow generation is not fully automatic as SevenBridges also uses graph based visual programming to generate workflows. Noteworthily, the open source CWL workflow executor and graphical composer, Rabix, is founded by SevenBridges[14].

RAWG solves the tedious process of using graph based visual programming by automating the links between tools. This is accomplished by using high quality logical programming which allows users to run workflows automatically and in a standardised fashion. In order to showcase the need for RAWG, this report will compare the outputs from a variety of common RNA-Seq workflows, illustrating the variability in results. To prove the ability of using this tool to make top quality scientific discoveries, we also ran three different analysis workflows, Differential Gene Expression (DGE), Differential Exon Expression (DEE) and Differential Isoform Expression (DIE), on neuroblastoma data provided by Dr Ximena Montano, the results will be shown in this report.

2 Methods

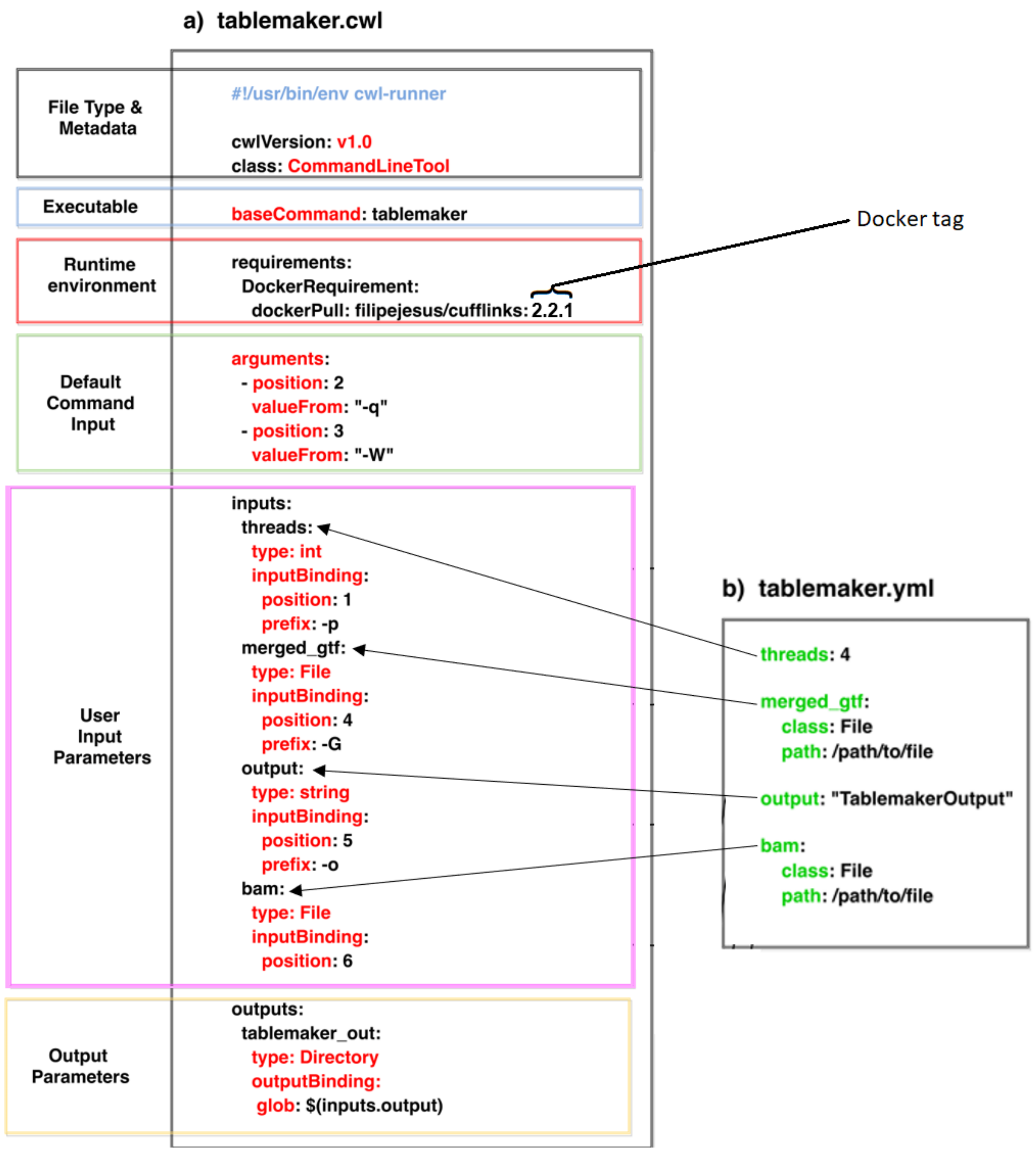
2.1 CWL Scripts *Work by FJ(70%), ZY(20%), APG(5%) and PHM(5%). Written by FJ.*

Our framework relies on CWL scripts and input files. In this section I explain the layout for both CWL tools and workflows and describe the tools for which the CWL scripts were developed.

2.1.1 CWL Command-line Tools

In total, 15 bioinformatics tools were wrapped into 31 CWL scripts (tools that requires multiple steps are wrapped into multiple CWL scripts). Each CWL script is linked to a docker image. Containers were used to ensure that all dependencies for the software are available thus making our software more portable. We tried to use docker images provided by Biocontainers[19] or the Genomic Paris Center[20] wherever possible, as these images are actively maintained. I also developed a few docker images to better fit our needs. Due to the large amount of scripts needed and the complexity of each tool, it became apparent early on that a consistent format (Figure 3) and standardised input/output names would be needed for our CWL scripts, in order to retain readability, allow for easy debugging and updates in the future. The following describes each section of a CWL-tool script:

- **File Type and Metadata** documents the CWL specification version and class of this script.
- **Executable** states the command that the script should run.
- **Runtime Environment** defines the necessary runtime environment of the tool, loads the docker images and specified modules. Note the docker tag which states a docker image version. This tag ensures the image is stable and should, theoretically, never change.
- **Default Command Inputs** is used to provide standard input parameters which are not user specific. These flags are used when the basecommand is executed in the command-line.
- **User Input Parameters** contains the variables which are populated by the input file information. In this section, information can be provided for each input, including name, type, position in the command-line script and prefix.
- **Output Parameters** specifies the name and type (can be files or directories) of outputs to save. The desired outputs are signalled by the outputBinding field.



```
#!/usr/bin/env cwl-runner

cwlVersion: v1.0
class: CommandLineTool

baseCommand: tablemaker

requirements:
  DockerRequirement:
    dockerPull: filipejesus/cufflinks:2.2.1

arguments:
  - position: 2
    valueFrom: "-q"
  - position: 3
    valueFrom: "-W"

inputs:
  threads:
    type: int
    inputBinding:
      position: 1
      prefix: -p
  merged_gtf:
    type: File
    inputBinding:
      position: 4
      prefix: -G
  output:
    type: string
    inputBinding:
      position: 5
      prefix: -o
  bam:
    type: File
    inputBinding:
      position: 6

outputs:
  tablemaker_out:
    type: Directory
    outputBinding:
     glob: $(inputs.output)
```

```
threads: 4

merged_gtf:
  class: File
  path: /path/to/file

output: "TablemakerOutput"

bam:
  class: File
  path: /path/to/file
```

Figure 3: The tablemaker CWL script and corresponding input file. Note the structured format of the script, each colour box represents a component of the script. The order and naming schemes are consistent throughout all our CWL scripts. This was adapted from a presentation given by Peter Amstutz[21].

Two of the top rated transcript aligners that were wrapped into CWL scripts are STAR[22] and HISAT2[23]. Each aligner uses a significantly different approach. STAR uses a Maximal Exact Match concept similar to those used in Mummer. STAR alignments are conducted using optimised suffix arrays that take a sequential alignment approach in which only unmapped regions of the reads are aligned to the genome. This makes STAR's algorithm more efficient. HISAT2, on the other hand, utilises genome indexes which are based on the FM-index and the Burrows-Wheeler transformation. HISAT2 uses a multi-step approach in which it first aligns to global indexes, followed by alignment extension and local alignments. This combination of steps gives HISAT2 high accuracy when aligning over complex structures like pseudogenes, splice junctions etc. Fortunately both aligners require similar inputs and both can be used in two steps, Indexing and Aligning. These aligners require from the user a genome annotation file (GTF), matching FASTA files and FASTQ files for each sample. In the CWL scripts we ensured that both aligners output results in SAM format in order to streamline downstream steps. On completion, the SAM files are sorted by genomic position and converted into BAM files, this conversion is done using a CWL wrapped SAMtools[24].

Assemblers can be grouped into our three analysis types. Importantly, while the uniqueness of each of these assemblers is apparent, we simplified the management of running their CWL scripts. Therefore, each of these assemblers run using similar inputs. The inputs are a combination of BAM files, annotation files, per sample library type information, metadata and FASTA files (tools specific input information can be found in the appendix). All assemblers output files containing feature count values for each sample. However, some assemblers necessitate post-processing to create a traditional count matrix for data analysis. Stringtie[25], Cufflinks[26], Salmon[27], and FeatureCounts[28] are all gene/transcript assemblers and thus can be used to quantify gene counts for DGE analysis. Cufflinks is an older transcript assembler, it utilises overlap graphs for transcript assembly and then quantifies the transcripts using maximum likelihood abundance estimation. Stringtie is a newer assembler, it builds splice graphs in which exons are represented as nodes and links to downstream exons are represented as edges, thus the paths through the graph represent all possible splice variants. For each variant a separate flow network is generated to estimate its expression level using a maximum flow algorithm.

Salmon and FeatureCounts are not traditional assemblers. Firstly, FeatureCounts can be thought of purely as a quantifier. Its main purpose is counting the number of reads which map to a feature within the user's annotation file. FeatureCounts uses a simplistic algorithm in which it defines a count as an overlap of one or more base-pairs between a read and its paired feature[28]. Whereas Salmon is an alignment-free mapper. Thus Salmon does not need STAR or HISTAT2 to be run prior to mapping, however it does require indexing much like those aligners. Salmon uses a lightweight mapping algorithm to map files directly to a transcriptome. Salmon then uses a two step approach, the first step is an online phase in which initial expression estimates and model parameters are calculated[27]. The second step is an offline phase in which the online estimates are used in a stochastic collapsed variational Bayesian inference algorithm to generate final expression estimates.

We have provided only a single assembler for DIE and DEE analysis, these are MISO[29] and HTSeq[30] respectively. HTSeq is a library which offers parsers for many RNA-Seq data formats. Its core component is the GenomicArray class which stores positional information on any genomic feature. A script provided by DEXSeq, dexseq_count.py, utilises the GenomicArray class to perform rapid alignments of reads to genomic features. Exon expression is measured by the number of reads overlapping the exon, similarly to FeatureCounts. MISO also calculates exon level expression however later associates the expression to a specific gene isomer. MISO predicts *de novo* splice junctions by spliced alignment to the genome. Specifically the estimation of isoform expression is calculated using the relative read density of spliced in and spliced out exons. This density is rescaled against the other possible isoforms. The density value is fed as input, along with other information such as read length, standard deviation and library size, into a Monte Carlo based inference technique to calculate isoform expression.

As with assemblers, analysis tools can also be grouped into DGE, DEE and DIE tools. The pairing of assemblers to analysis tools is complex, due to file types and statistical constraints, Figure 4 shows all possible connections between tools. DESeq2[31], CuffDiff[32], Ballgown[33] and edgeR[34] are the available DGE tools, DEXSeq[35] is the available DEE tool and MISO Compare[29] is the available DIE tool. Each tool uses a unique normalisation method for differential gene expression analysis aside from DEXSeq which is a variant of DESeq2 modified for exon analysis. All tools correct for multiple testing using the Benjamini–Hochberg procedure for control of the false discovery rate (FDR)[36].

The post-analysis element of our framework consists of a FGSEA[37] and hypergeometric mean[38]

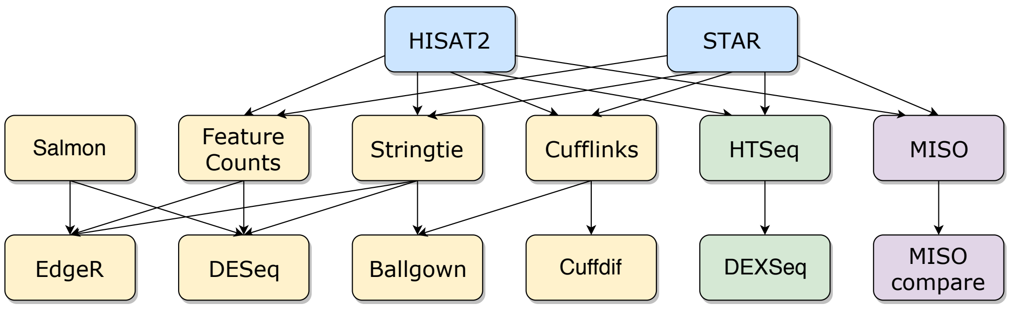


Figure 4: Flowchart of all possible workflows. All possible links between tools has been shown. Assembler are shown in blue, tools for DGE analysis shown in yellow, DEE tools shown in green and DIE shown in purple. The diagram has been split into three rows, the first shows all aligner, the second shows all assemblers and the last shows all analysis tools.

CWL scripts, and a data visualisation portal which is described later. FGSEA is a pre-ranked gene set enrichment protocol which produces enrichment scores that are equivalent to GSEA developed by the Broad Institute. Traditional gene set enrichment analysis, such as GSEA, looks at a ranked list of genes and identifies sets of genes that are on one side of the ranked list more so than would be expected by chance[39]. FGSEA was chosen due to its optimised gene set sampling method which produces near identical results to GSEA in less computational time[37]. Hypergeometric mean is an over-representation analysis which utilises the Fisher's exact test-like statistic. Essentially, the technique uses a contingency table of the number of differentially expressed and not differentially expressed genes within and outside the gene set. This allows the identification of gene sets which are over-represented by differentially expressed genes. Both scripts take two inputs, DGE analysis results and a gene sets file in long-form format where gene ids are contained within the first column and pathways ids in the second.

2.1.2 Workflows

Although workflows will be generated on-the-fly in the end product, we hand coded six workflows to study the interconnection of the tools which will provide preliminary ideas for the workflow generator (Figure 5). They also allow us to test all main and utility tools in a workflow like scenario while testing CWL's ability to run branched workflows. CWL workflow scripts have a standardised format, this format is similar to the CWL tool scripts, except the definition of tool execution occurs in the "Workflow steps" phase. Figure 6 demonstrates the format of CWL workflow scripts. Each step within "Workflow steps" can be separated into three subsections:

- **run** - specifies the CWL tool script to use in the step.
- **in** - list inputs for the CWL tool script.
- **out** - specify the output desired. These can be used as inputs in downstream tools.

To ensure a coherent design pattern was used through-out our development process, continuous testing was implemented with Travis CI, which conducts automated testing on every iteration of our product. This would alert us if any updates break our existing tools. The results from each CWL workflow were compared to results of the same workflow when run in a shell script. Testing against workflows written in traditional shell script is important as it ensures that our workflows perform as users of traditional methods would expect. In the same vein, their outputs were also used to compare against results from automatically generated workflows.

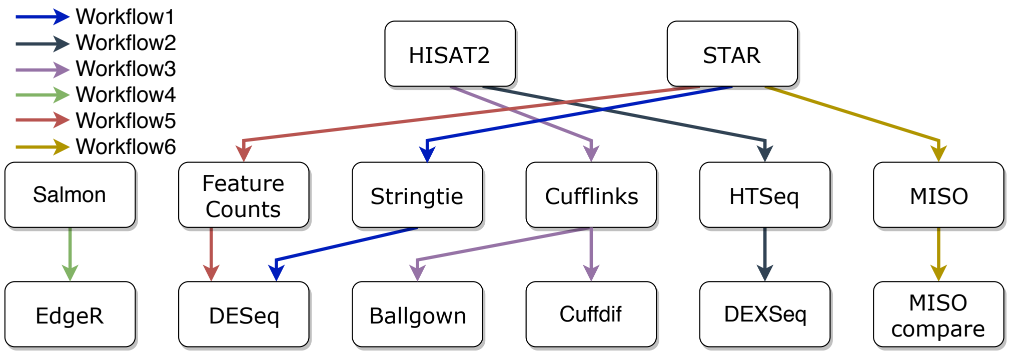
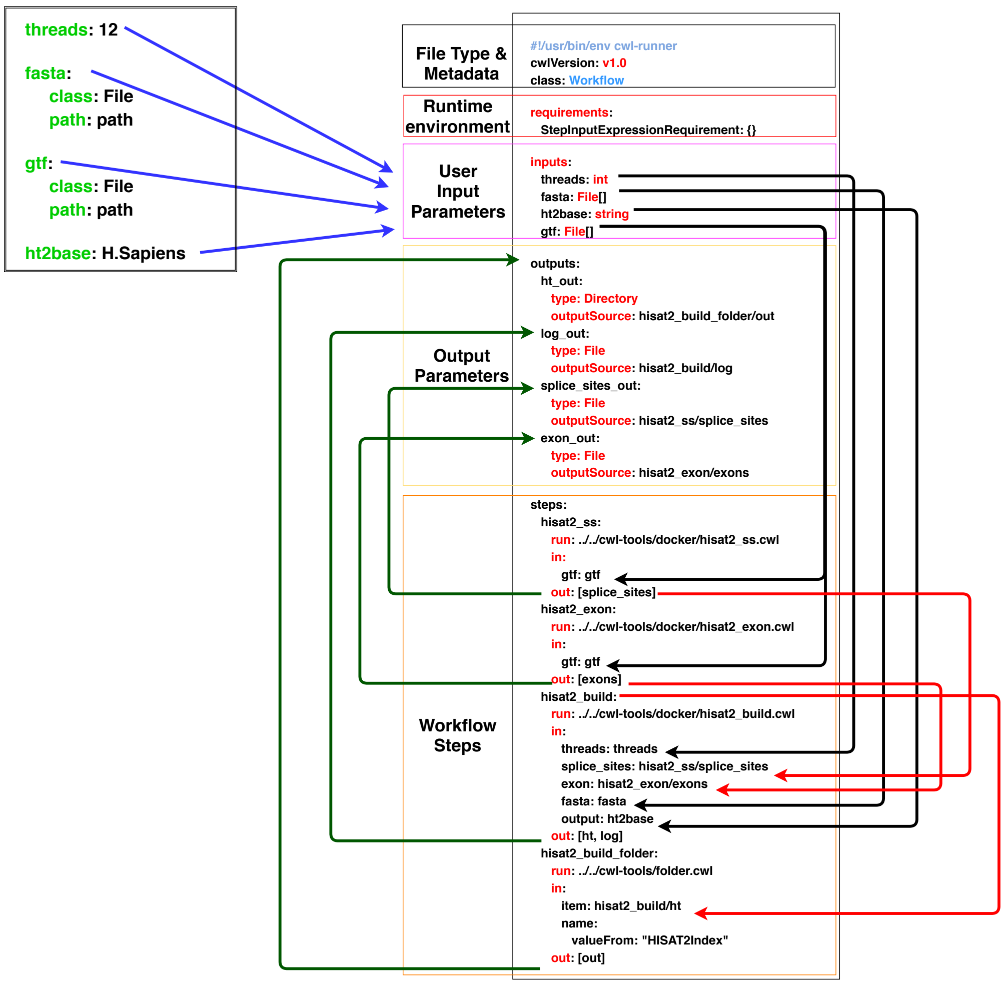


Figure 5: Visualisation of the six workflows which were manually created. Each workflow is represented as a path with edges of unique colour. As can be seen the workflows cover all the tools and workflow 3 is a branched workflow with both Ballgown and Cuffdiff as analysis steps.



```
threads: 12

fasta:
    class: File
    path: path

gtf:
    class: File
    path: path

ht2base: H.Sapiens
```

```
#!/usr/bin/env cwl-runner
cwlVersion: v1.0
class: Workflow
requirements:
  StepInputExpressionRequirement: {}
inputs:
  threads: int
  fasta: File[]
  ht2base: string
  gtf: File[]
outputs:
  ht_out:
    type: Directory
    outputSource: hisat2_build_folder/out
  log_out:
    type: File
    outputSource: hisat2_build/log
  splice_sites_out:
    type: File
    outputSource: hisat2_ss/splice_sites
  exon_out:
    type: File
    outputSource: hisat2_exon/exons
steps:
  hisat2_ss:
    run: ../../cwl-tools/docker/hisat2_ss.cwl
    in:
      gtf: gtf
    out: [splice_sites]
  hisat2_exon:
    run: ../../cwl-tools/docker/hisat2_exon.cwl
    in:
      gtf: gtf
    out: [exons]
  hisat2_build:
    run: ../../cwl-tools/docker/hisat2_build.cwl
    in:
      threads: threads
      splice_sites: hisat2_ss/splice_sites
      exon: hisat2_exon/exons
      fasta: fasta
      output: ht2base
    out: [ht, log]
  hisat2_build_folder:
    run: ../../cwl-tools/folder.cwl
    in:
      item: hisat2_build/ht
      name:
        valueFrom: "HISAT2Index"
    out: [out]
```

Figure 6: Visualisation of the workflow script format. Boxes show unique sections of the script, format is similar to CWL scripts but contains "Workflow steps". Inputs are used in the "Workflow steps", black arrows. Outputs of each step can be used for downstream tools, red arrows. Only "out" variables which are linked to an "output parameter" are returned to the user, green arrows.

2.2 Database design *Work by APG(25%), PHM(25%), FJ(25%) and ZY(25%). Written by PHM.*

The database was at the centre of the application and acted as a link between the user defined workflows and the workflow scripts that would be generated. As a consequence, its design was a highly collaborative process and the responsibility for its design was shared equally among us.

To provide the necessary functionality the database would need to store genome data uploaded by the user and map these to appropriate workflows. The architecture would need to be flexible enough to deal with the multiple user defined parameters whilst maintaining data integrity.

The key advantage of RAWG is its ability to dynamically generate parallel pipelines. Hence we anticipate that users will want to compare multiple workflows and within each workflow they may

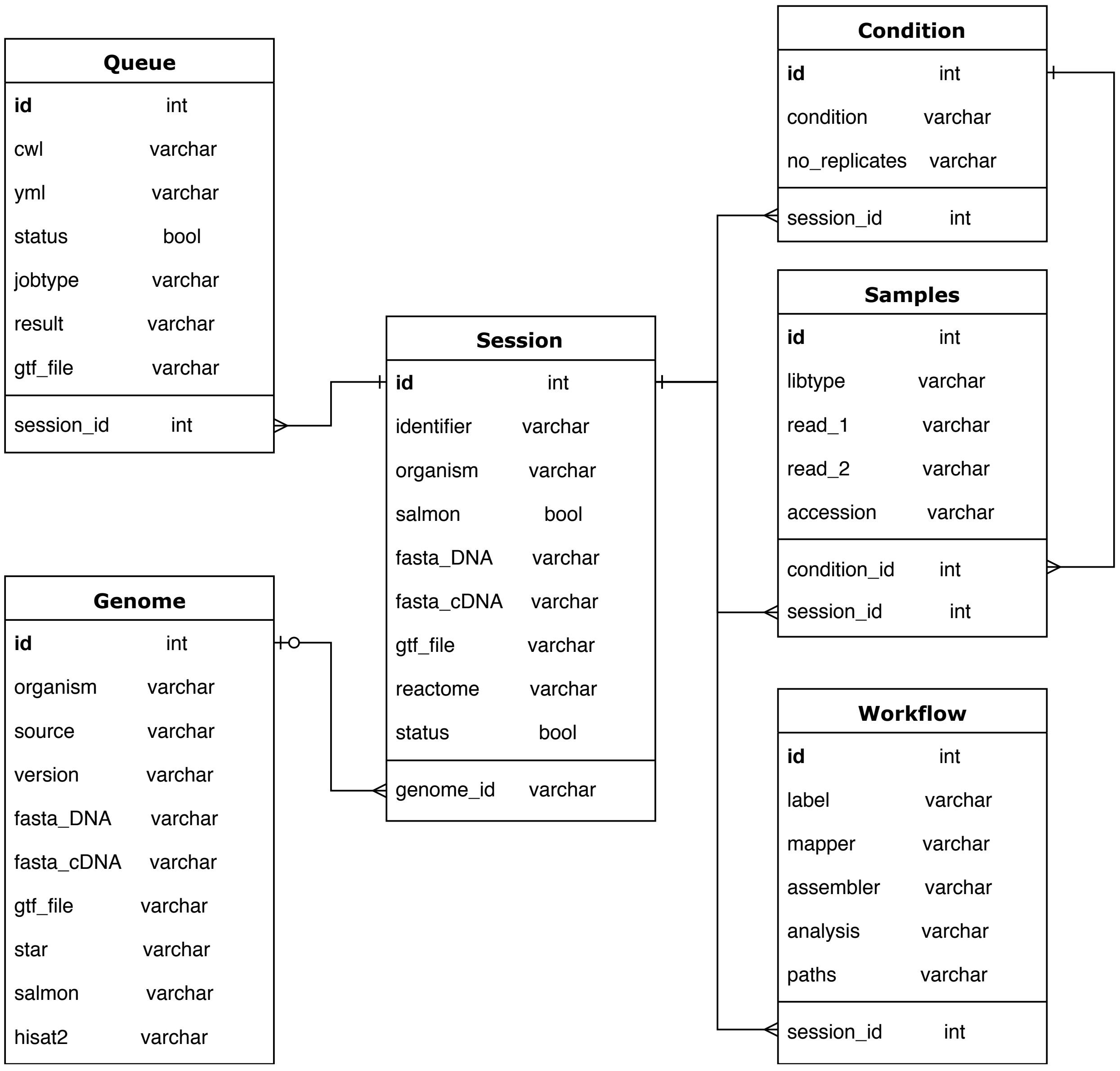


Figure 7: Overview of the database schema. Session Table is linked to Condition, Samples, Workflow and Queue Tables by a one to many relationship. Genome is linked to Session by a non-mandatory one to many relationship. Condition is linked to Samples by a one to many relationship.

want to include multiple samples of varying tissue type. To represent this in a database, we decided to use session as a way of partitioning the users without having to manage user accounts. The overview of our database architecture is laid out in Figure 7.

The Session table was initialised with an auto-incrementing primary key which could be used to link a session instance to RNA-Seq data and selected workflows. In addition, we decided to add an unique identifier generated using Python's UUID module. This enabled the creation of session specific directories to store uploaded data. A character field was also used to allow the user to label organism of origin for the sequence data. Character fields were defined for DNA FASTA, cDNA FASTA and GTF files which would be used to store paths to the uploaded data. The cDNA FASTA file would only be required if the user specified that Salmon would be required for down-steam analysis. These reference genome and annotation files would be required for the computationally intensive genome indexing step in the first part of the alignment process. An integer field, status, was defined and would be set to 1 when the user had submitted their workflows.

Given that one of the most computationally intensive steps in our pipeline is genome indexing and that this step needs only to be run once for each pair of reference genome and annotation files, it would benefit the user to have access to commonly studied organisms. In this way, we could anticipate the computational requirement and preempt the demand.

To accommodate this, we initialised a Genome table with the same fields as were defined in the Session table plus the source and version number of the reference genome. The auto-incrementing primary key in the Genome table was referenced by a genome_id foreign key field in the Session table. In this way, we created a non-mandatory relationship between the session instance and one of the pre-indexed genomes in our Genome table. If the user selected a pre-indexed genome then the redundant values in the Session table would be set to null as no uploads would be required since the workflow scripts could be run on the pre-indexed files.

Both the Session and Genome tables were initialised with unique field names and took single data type as input. The values were themselves single elements rather than lists or tuples thus satisfying the condition for normalisation under the first normal form. There were no partial dependencies as unique auto-incrementing integers were used as the primary keys and hence we did not have to rely on composite keys for data retrieval.

Having defined the Session table we then needed to map the uploaded RNA-Seq files to the appropriate session instance. This posed a challenge because the user defined parameters included the tissue condition of the sample, the number of replicates from each tissue and the library type of the reads which could be paired or single ended. To accommodate this we decided to split the samples table into a Condition and Samples table. Each of these were initialised with a foreign key "session_id" which were both linked to the primary key of the session table to associate them with the correct session instance.

The Condition table was initialised with an auto-incrementing primary key field, a condition field and a number of replicates field. A user could then define the number of sample replicates associated with each tissue condition. The Samples table stored the library type (paired or single ended reads) along with read_1 and read_2 fields which would store the file paths of the user uploaded samples. If single read library type was selected, the read_2 field would be set to null. This resulted in some data redundancy which was limited to the read_2 field. This was preferable to storing each read as a separate instance which would have meant redundancy in more than one field.

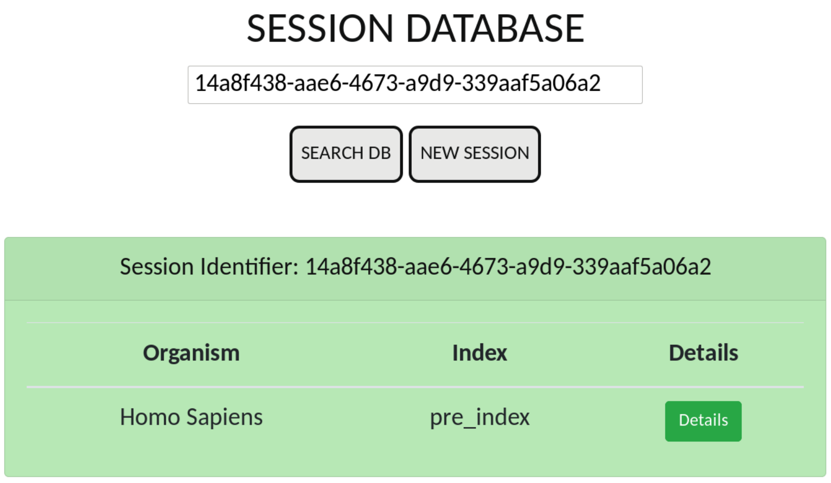
By normalising in this way we reduced data redundancy as the sample condition did not need to be included in each row of the Samples table. This prevents insertion anomaly where by the user must redefine the condition when uploading each sample. Deletion and update anomaly are also prevented since the user can remove all instances from the Sample table without losing the associated tissue condition. This normalisation procedure had the additional benefit of reducing the chance that a user would mislabel a sample condition, as they only had to define each condition once. Similarly, separating out the number of replicates and having the user define them explicitly means we could perform a front-end validation to ensure the number of submitted samples matched the number of replicates defined in the Condition table.

To link the submitted workflows to the session instance, we defined a Workflow table which was linked by a foreign key "session_id" to the primary key in the Session table. A label field allowed users to name their workflows for identification during visualisation. The mapper, assembler and analysis fields were then populated by user selection from our pre-defined list of tools. The paths field, set to null initially, would be populated with file paths to the csv files generated by those selected workflows.

2.3 Front-end *Work by PHM. Written by PHM.*

We decided to use the Django web framework (version 2.1.5) for the development of our website for which I took primary responsibility. The Django paradigm is that database tables are defined in `models.py` which the user can then interact with via a html template. This suited our use case since users would need to upload files and make queries through a web interface.

Primarily, the website needed to provide an intuitive way for users to upload files, select workflows and then visualise their analyses. It also needed to perform this in a session isolated environment to prevent users from manipulating each others files or workflows. The home page, as shown in Figure 8, presented users with the choice to start a new session or to find a previously submitted one using the unique session identifier.

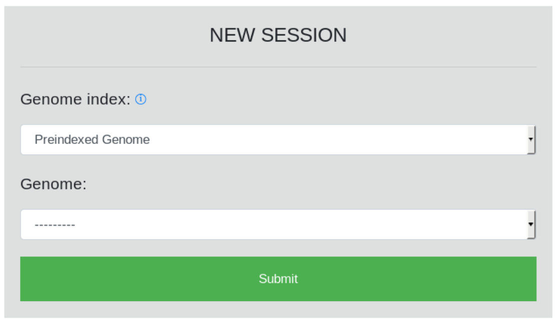


Session Identifier: 14a8f438-aae6-4673-a9d9-339aaf5a06a2

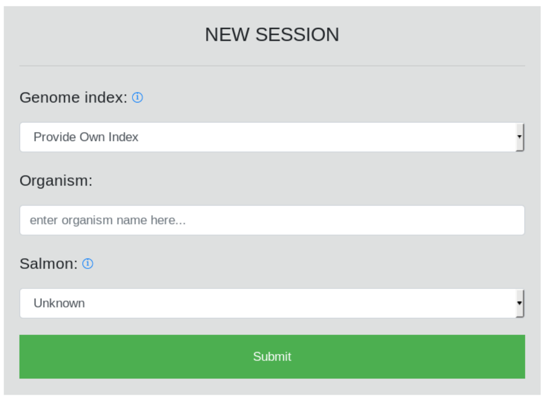
Organism	Index	Details
Homo Sapiens	pre_index	Details

Figure 8: Website home page. User has performed a successful lookup of a session instance and can use the Details button to navigate to the dashboard.

Starting a new session would navigate the user to the session_create page where they could define the parameters associated with a new session as shown in Figure 9. JavaScript was used to build form fields dynamically depending on the user choices.



(a)



(b)

Figure 9: On initialising a new session the user has the option to use a pre-indexed genome (a) or to provide their own (b).

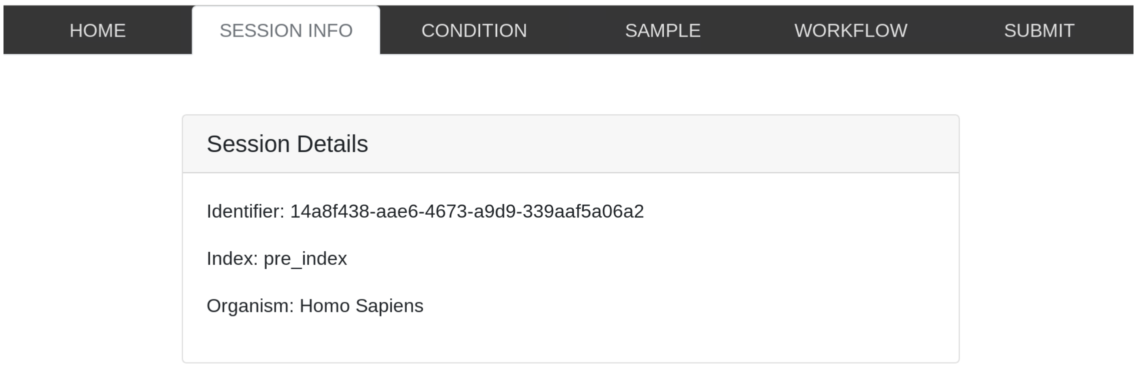
Selecting to use a pre-indexed genome, Figure 9a, would allow users to select from our list of pre-defined organisms whereas choosing to upload their own genome, Figure 9b, would begin building an alternate form that would allow users to upload necessary genome files.

Upon form submission, this data would be saved to the new instance in the Session table and the user would be forwarded to the session_detail page. However a user could also locate a previous session from the home page by inputting their unique identifier as seen in Figure 8. I used the identifier generated by the UUID module to perform this lookup as it was unique to each session and was more secure than looking up by the session's primary key. If the identifier matched the full character string in the database the user would be provided a link to the session_detail page. Otherwise they would be informed that there was no session associated with that identifier. This was an important feature to add since some of the workflows would take hours to run. Expecting a user to keep the browser tab open for the entire duration would be unreasonable, severely limiting RAWG's usability. With this feature, a user could return to their session dashboard at any time to check the progress of their analysis.

The session_detail page was designed to be the primary dashboard from which a user could complete all the uploads and selections that would be required for the downstream analysis. The URL for this page was constructed by appending the unique session identifier to the hostname/session_detail/ producing hostname/session_detail/unique-identifier. This allowed me to access the unique identifier within the `views.py` file on the server-side, which I used to query the Session table and return only uploads and workflows that were associated with the correct session instance. Any activity by the user on the session_detail page would then further append a function call to the URL producing hostname/session_detail/unique-identifier/an-example-function. This provided a general mechanism to execute functions for data linked to a unique session instance.

The design of the dashboard reflected the underlying database structure with a tab for each table in the database across the top of the page as shown in Figure 10a. To keep the user experience intuitive the functionality to add workflows depended on the user having added samples, and adding samples depended on them having defined the conditions. In this way the user was guided through the upload and selection process. The first tab, "Session Info" gave the user their unique identifier along with their index choice and organism. In the second tab shown in Figure 10d, users could define the conditions associated with their RNA-seq samples using the condition_create form (Figure 10b and 10c). Submission of this form returns the user to the dashboard where they would now be able to upload samples, amend the condition information or delete it entirely as seen in Figure 10d.

Structuring the uploads in this way meant any samples would be automatically associated with a predefined condition. This minimised the risk that a user would mislabel a sample because each unique tissue condition would be explicitly defined only once.



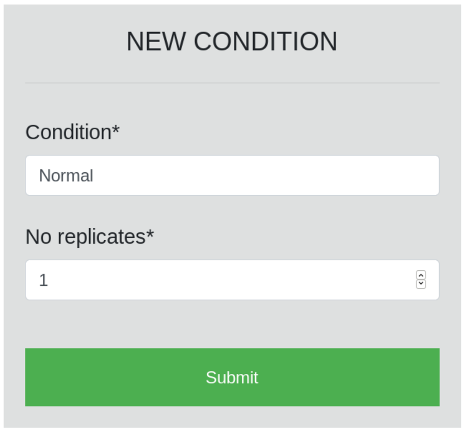
Session Details

Identifier: 14a8f438-aae6-4673-a9d9-339aaf5a06a2

Index: pre_index

Organism: Homo Sapiens

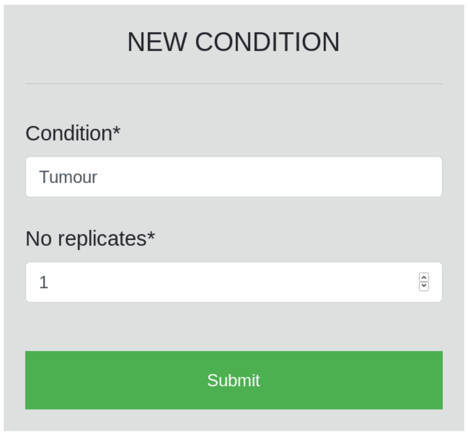
(a) Session Info Tab provides overview of the session details.



Condition*: Normal

No replicates*: 1

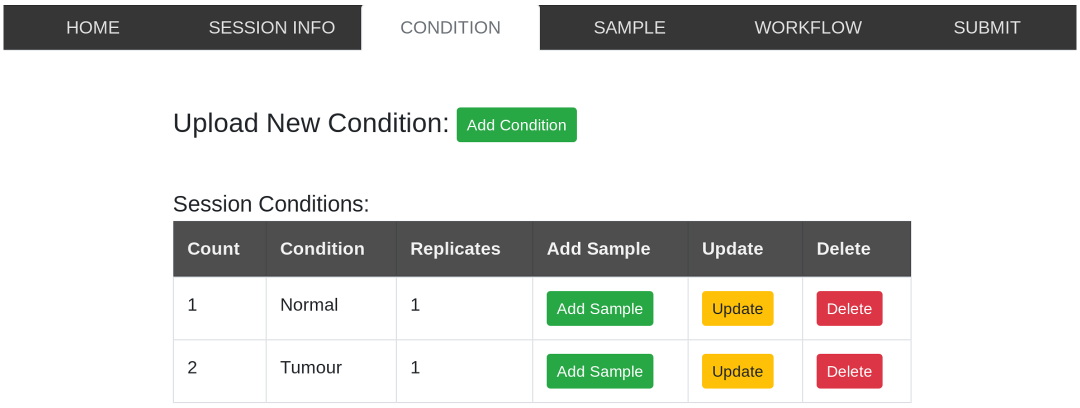
(b) Normal tissue condition



Condition*: Tumour

No replicates*: 1

(c) Tumour tissue condition



Upload New Condition:

Session Conditions:

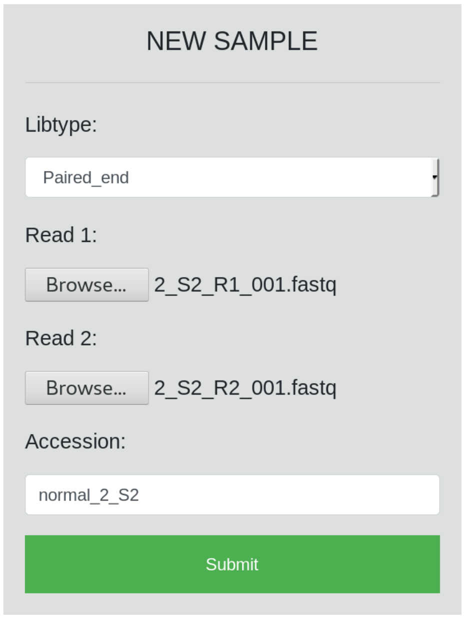
Count	Condition	Replicates	Add Sample	Update	Delete
1	Normal	1	Add Sample	Update	Delete
2	Tumour	1	Add Sample	Update	Delete

(d) Populated Condition Tab

Figure 10: Overview of condition definition. The user has defined two conditions, Normal and Tumour and can now upload samples associated with these conditions.

As with the new session form, the samples upload form seen in Figure 11a and 11b used JavaScript to generate the appropriate form fields as the user specified their sample library type. A check was performed at each sample upload event to ensure that the user had not submitted more samples than they had specified in the conditions form. If a user decided to relabel their conditions following sample uploads the samples would automatically be reassigned to the new condition because of the normalisation steps we had taken during database design.

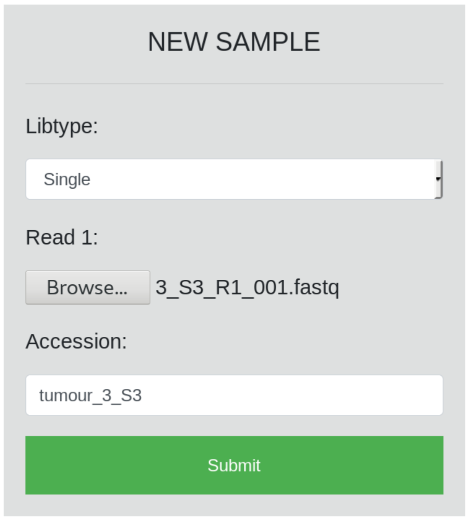
In the third tab "Samples", a user could review their RNA-Seq uploads and correct any errors by updating their samples. As discussed in database design, read_2 was not a mandatory field, allowing a user to upload single or paired end reads and have them stored in a single instance of the Samples table. A drawback of this method was that back-end database validation, ensuring that both fields had been completed was not possible. To avoid inconsistent data where a user had specified paired end reads but only submitted a single file it was necessary to perform client-side form validation. JavaScript was used to check whether the user had selected paired end reads and provid user feedback if they had not uploaded a second file.



NEW SAMPLE

Libtype:	Paired_end
Read 1:	Browse... 2_S2_R1_001.fastq
Read 2:	Browse... 2_S2_R2_001.fastq
Accession:	normal_2_S2

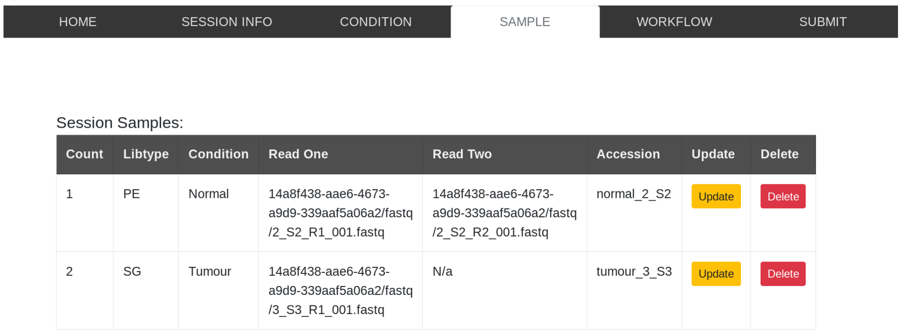
(a) Paired End read from normal tissue



NEW SAMPLE

Libtype:	Single
Read 1:	Browse... 3_S3_R1_001.fastq
Accession:	tumour_3_S3

(b) Single end read from tumour tissue



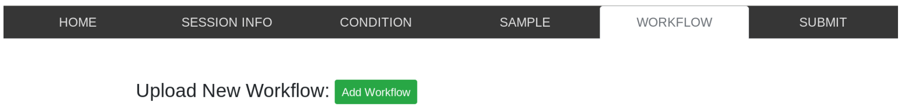
Session Samples:

Count	Libtype	Condition	Read One	Read Two	Accession	Update	Delete
1	PE	Normal	14a8f438-aae6-4673-a9d9-339aaf5a06a2/fastq/2_S2_R1_001.fastq	14a8f438-aae6-4673-a9d9-339aaf5a06a2/fastq/2_S2_R2_001.fastq	normal_2_S2	Update	Delete
2	SG	Tumour	14a8f438-aae6-4673-a9d9-339aaf5a06a2/fastq/3_S3_R1_001.fastq	N/a	tumour_3_S3	Update	Delete

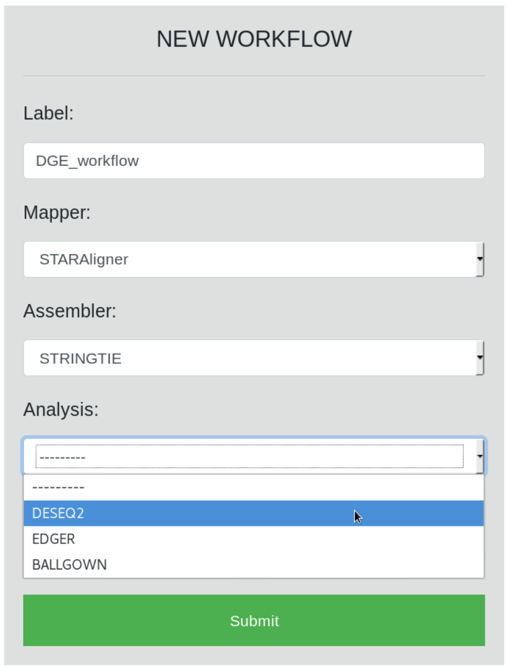
(c) Samples Tab view with associated uploads

Figure 11: Overview of Sample creation. User has uploaded a paired end and a single ended samples

Once the user had uploaded their samples, they would be able to select their custom pipelines from the workflows tab (Figure 12a). The drop down form fields were pre-populated with RAWG provided tools. However the options would need to be dynamically filtered depending on user input given that the choice of mapper would restrict the choice of assembler which would in turn restrict the choice of downstream analyses.



(a) Workflow Tab



Label: DGE_workflow

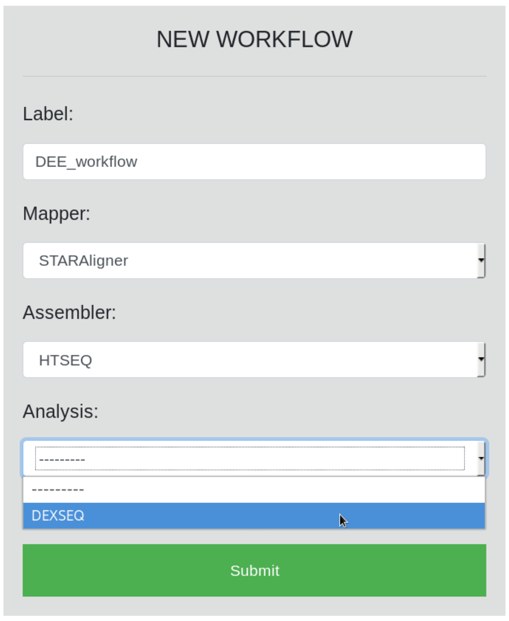
Mapper: STARAligner

Assembler: STRINGTIE

Analysis: ---------

- ---------
- DESEQ2
- EDGER
- BALLGOWN

(b) DGE Workflow using StringTie for assembly



Label: DEE_workflow

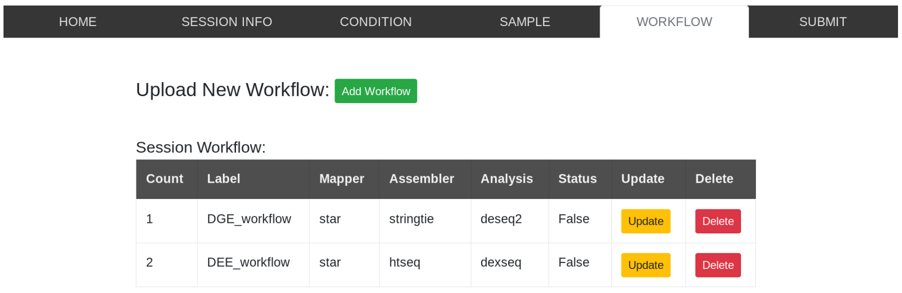
Mapper: STARAligner

Assembler: HTSEQ

Analysis: ---------

- ---------
- DEXSEQ

(c) DEE Workflow using HTSeq for assembly



Session Workflow:

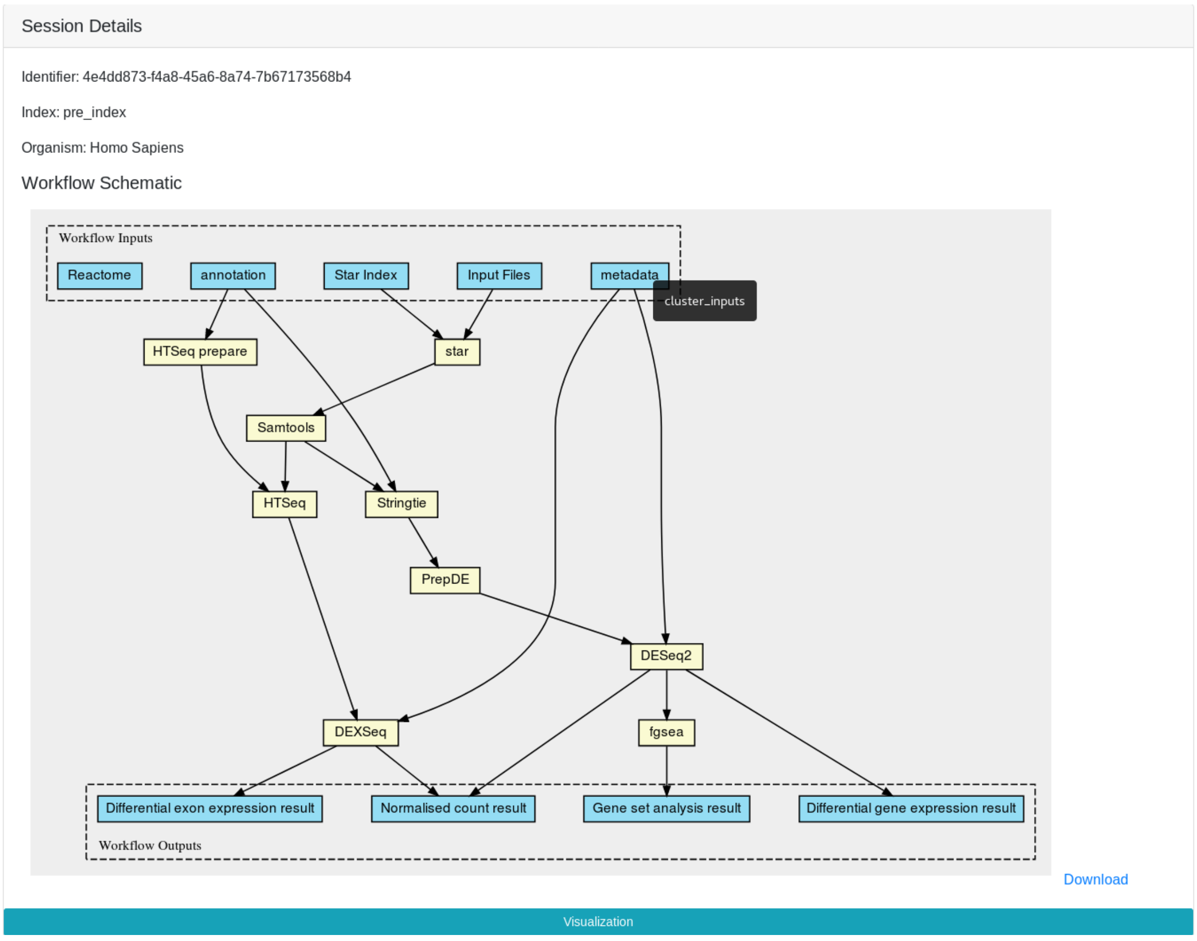
Count	Label	Mapper	Assembler	Analysis	Status	Update	Delete
1	DGE_workflow	star	stringtie	deseq2	False	Update	Delete
2	DEE_workflow	star	htseq	dexseq	False	Update	Delete

(d) Workflow tab view showing DGE and DEE selected workflows.

Figure 12: Overview of Workflow selection. User has selected two custom workflows using StringTie and HTSeq

This was solved using AJAX to asynchronously pass the users selected mapper to an assembler dictionary which would return all permitted assemblers for the selected mapper. This was repeated for the analysis drop down as shown in Figures 12b and 12c.

Having selected their desired workflows the user would then be able to review and alter them in the Workflows tab as shown in Figure 12d before submitting their custom pipeline for analysis. Submission would automatically generate an SVG image showing an overview of the selected workflows which the user would be able to download, see Figure 13.


Session Details

Identifier: 4e4dd873-f4a8-45a6-8a74-7b67173568b4

Index: pre_index

Organism: Homo Sapiens

Workflow Schematic

Figure 13: Dynamically generated SVG summarising user pipeline

2.4 Server-side *Work by ZY(80%) and APG(20%). Written by ZY.*

A database-centric approach was used through out the architectural design of the project's server-side functionalities. We utilised database entries to keep track of all the user-uploaded files, analyses result files, as well as a status column to track the state of sessions and jobs. We coded Python classes that can extract relevant records from database and resolve all the steps that the final workflow need to take while common steps are shared to avoid duplications. The steps are then translated to CWL workflow codes and submitted to a job queue. I implemented a sequential job scheduler based on the Queue table described in Section 2.2 to run the generated workflows so multiple sessions can be submitted as the same time. With one central database to keep track of the data pipeline progress through out, we can make sure that all parts of our framework work in harmony. The overall design of the server-side functions is summarised in Figure 14 and the general organisation of scripts is shown in Figure 15.

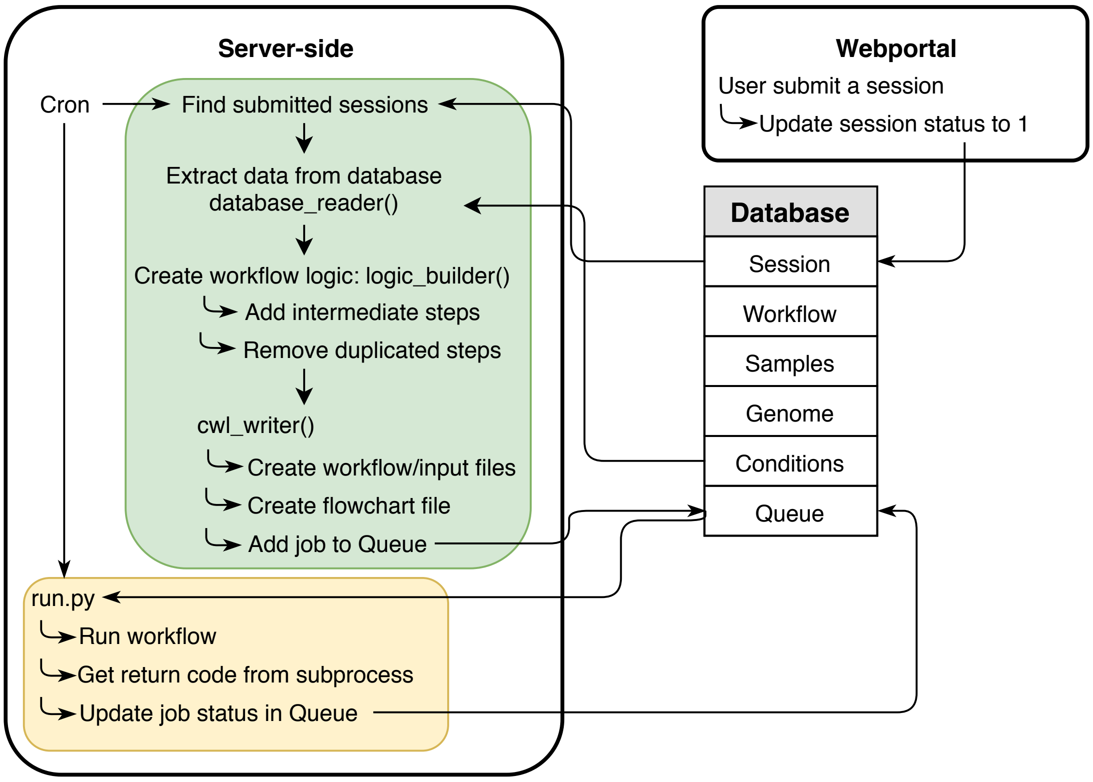


Figure 14: Flowchart illustrates the steps taken in the server-side after job submissions.

```
Project Root
└─ backend_scripts
   ├─ programs.py
   │  └─ class cwl_writer()
   ├─ classes.py
   │  ├─ class database_reader()
   │  └─ class logic_builder()
   ├─ cwl_creater.py
   ├─ run.py
   └─ programs_connections.csv
```

Figure 15: Scripts written for the server-side functionality and the defined Python classes.

2.4.1 Workflow Generation

Three major steps are involved in programmatically generating CWL workflows:

1. extracting user defined workflows from Workflow table
2. resolve any intermediate steps between tools and remove duplicated steps

3. add CWL compliant descriptions for each step to the final workflow

I used SQLAlchemy as the object-relational mapper (ORM) to map database tables to Python classes which provides a pythonic method of querying SQL database. SQLAlchemy's syntax is database independent which means virtually no change was needed when switching from a local development database (eg. SQLite) to a remote production database (eg. MySQL).

Once `cwl_creator.py` found submitted sessions, the script will extract relevant information from the database and save them into a database_reader object (an instance of `database_reader` class). A logic_builder object is initiated and the workflow steps are created by passing the database reader object to the logic_builder. A cwl_writer object is initiated with the database_reader object and by passing the logic_builder object to the cwl_writer object, the write_workflow method of cwl_writer loops through all the unique steps constructed by the logic_builder and write each step as a dictionary entry in the cwl_writer's cwl_workflow attribute (Figure 14).

Resolve unique steps

To illustrate the process taken by `logic_builder` to generate the unique steps for the workflow, I will use Table 2 as an example.

id	session_id	label	mapper	assembler	analysis
1	1	wf1	star	stringtie	deseq2
2	1	wf2	star	featurecounts	deseq2
3	1	wf3	star	stringtie	ballgown

Table 2: An example Workflow table with duplicated steps

When Session 1 is submitted, workflows that are linked to Session 1 are read into a nested list followed by adding the intermediate steps that connect the tools, as shown below:

```
[['star', 'samtools', 'stringtie', 'prepde', 'deseq2'],
 ['star', 'samtools', 'featurecounts', 'deseq2'],
 ['star', 'samtools', 'stringtie', 'ballgown']]
```

	star	stringtie	featurecounts	deseq2	ballgown
star	-1	samtools	samtools	-1	-1
stringtie	-1	-1	-1	prepde	0
featurecounts	-1	-1	-1	0	-1
deseq2	-1	-1	-1	-1	-1
ballgown	-1	-1	-1	-1	-1

Table 3: An example of programs_connection table

The information regarding connection of tools (`programs_connections.csv`) is formatted as a table where the rows are the previous programs and the columns are the following programs (Table 3). The elements in the table denotes intermediate steps needed to chain these two programs. Additionally, 0 means no intermediate steps needed and -1 means unsupported connection.

After inserting intermediate steps, the list of steps are concatenated with previous steps to create an unique string for each unique step, as shown below:

```
[['star', 'star_samtools', 'star_samtools_stringtie', 'star_samtools_stringtie_prepde',
  'star_samtools_stringtie_prepde_deseq2'],
 ['star', 'star_samtools', 'star_samtools_featurecounts',
  'star_samtools_featurecounts_deseq2'],
 ['star', 'star_samtools', 'star_samtools_stringtie',
  'star_samtools_stringtie_ballgown']]
```

The list is flattened and converted to a set which means duplicated items are removed. The resulting set is then sorted alphabetically which converts the set back to a list (Listing 1).

```
['star', 'star_samtools', 'star_samtools_featurecounts',
 'star_samtools_featurecounts_deseq2', 'star_samtools_stringtie',
 'star_samtools_stringtie_ballgown', 'star_samtools_stringtie_prepde',
 'star_samtools_stringtie_prepde_deseq2']
```

Listing 1: The final unique steps list passed to cwl_writer

It is important to note that this sorted list contains all the unique steps needed for the three defined workflows, and the ordering means all steps' previous analyses are called before the later ones, which is necessary for the flowchart generation procedure.

Generate CWL workflow

Following our preliminary study on CWL workflows, we learnt that CWL files are YAML formatted files in essence which can be suitably represented by Python dictionary objects. A few Python packages exist to aid the programmatic generation of CWL files, however they either lack features that are crucial to our workflows or are very verbose. In the end, we decided to create a cwl_writer class which directly add steps to a dictionary as key-value pairs of strings.

This final list (shown in Lisiting 1) is passed to cwl_writer's write_workflow method which uses a loop to go though all the strings in the list in order. The string is split by underscore and the last element is used as the name to call the respective method in the class. In addition, three attributes of the class are modified before the method is called (Listing 2).

As the psudocode of method shown in Listing 3, each method mainly have five parts.

1. **inputs** which specifies any inputs needed by this step for the workflow. The value or path for the input is added to the workflow input dictionary as well.

```
def write_workflow(self, steps):
    for step in steps: # step = 'star_samtools'
        # a list of tools lead to this step eg. ['star', 'samtools']
        self.name_list = step.split("_")
        # an unique name for the current step eg. "star_samtools"
        self.name = step
        # the previous step's name eg. "star"
        self.previous_name = "_".join(self.name_list[:-1])
        # call self.samtools method
        getattr(cwl_writer, self.name_list[-1])(self)
```

Listing 2: Code snippet from cwl_writer's write_workflow method

2. **outputs** specifies the directories that the workflow should collect to save as output from this step.

3. **steps** specifies the CWL tool to run for this step, as well as the inputs and outputs configuration for this tool.

4. **foldering** is an utility step which collects the outputs from the tool and package them into a directory object. This step is to help organising output files.

5. **graph** is the step that add nodes and edges to the DOT graph object. This step adds a node named by the current step name and connect it to the previous step hence it is important to have the previous step defined as a node beforehand. This is the reason that the final list need to be ordered alphabetically.

The first three parts closely mirror the workflow step structure shown in Figure 6. The main part of each method is to add the step that describes the necessary inputs for the tool and the outputs to collect after the tool finishes running. It should be noted that some tools require multiple steps in the workflow where each step is the tool ran on one sample, such as the aligners. This required additional considerations when designing CWL wrappers for tools and the workflow generator. For tools that are R scripts in essence, all files are passed to the R script by CWL and the script will conduct analysis on each one of the files. For the stand alone tools (eg. stringtie), we loop over all the files in cwl_writer and create an individual step for each sample. For tools that belong to the differential expression category, an additional step is taken in the class method to add paths of the result files to the database. This is done to simplify the visualisation part's data retrieval process and fit into our database-centric philosophy. After every steps are generated, the workflow dictionary object is written to a CWL file following YAML format, so is the workflow input object.

```
def a_method(self):
    # add workflow inputs
    self.cwl_workflow["inputs"][...] = ...
    self.cwl_input[...] = ...
    # add workflow outputs
    self.cwl_workflow["outputs"][self.name + "_out"] = ...
    # add step to workflow
    self.cwl_workflow["steps"][self.name] = {
        "run": "path/to/cwl/scripts",
        "in" {
            # inputs for CWL wrapped tool as a dictionary
        }
        "out": ["output"] # outputs from tool
    }
    # foldering
    self.cwl_workflow["steps"][self.name + "_folder"] = ...
    # graph
    self.graph.add_node(pydot.Node(self.name, label="..."))
    self.add_edge()
```

Listing 3: Psudocode illustrates the typical construct of a method to add parts of a workflow. A full example is included in Appendix: Listing 4.

The workflow file and the input file's paths are then added to the Queue table for it to be run by the server.

As described before, the users have the options of providing their customised genome files and annotation files which means an extra indexing step is required if the "genome_index" field in the Session table equals "user_provided". In the write_workflow method, if indexing step is deemed to be necessary, a create_indexing method will be called which will write a workflow to generate the necessary index files. This indexing workflow is generated in a similar fashion as the analysis workflow. After the indexing workflow is created, the indexing job will be added to the Queue table. It is important that the indexing job is added before the analysis job otherwise the analysis job would fail because the indexing files will not exist.

2.4.2 Flowchart Generation

CWL provides an online viewer to explore publicly accessible CWL workflows on code hosting websites[40]. We liked the concept of a flowchart that illustrates the steps taken in a workflow hence decided to add workflow's flowchart to the webportal. Although CWL Viewer is open source, it is designed to be a stand alone website and the code could not be integrated into our codebase easily. Similarly, the reference CWL runner, cwltool, can generate a DOT (a graph description language) formatted file from a CWL workflow file. The DOT file is then rendered into standard images by GraphViz software. However, cwltool only generates a bare minimum DOT file which does not contain the features that make CWL Viewer stand out, such as grouping workflow inputs and outputs, displays data flow between steps and clean visual formatting. Therefore, I decided to add a functionality into cwl_writer that generate a visually pleasing flowchart while the workflow is generated.

Pydot was used in Python to provide an object-orientated way of storing the graph object. The additional code in tools' method first add a node of the current step to the graph then add an edge between the previous step and the current step. Some formatting code was used to match the style of the generated DOT file to CWL Viewer's style. After creating the DOT file, a subprocess was called to run GraphViz and render the DOT file into a SVG file for displaying on the webportal (an example is shown in Figure 13).

2.4.3 Job Scheduling and Queueing

It is important to have a scheduling system to avoid too many workflows run at the same time which degrades system performance. I decided to code a rudimentary queueing system that only one workflow will be running on the testing server at the same time. The `run.py` script uses Python's Subprocess module to open a system process that runs the cwltool command and wait until the workflow is finished. The finished job's entry in the Queue table will be updated based on the return code received by the subprocess module. Several additional options are specified when running the cwltool command. The --provenance option saves the workflow's provenance metadata as a Research Object to a specified path. The --timestamp and --debug options add timestamps and additional details to the log file, respectively.

A file-based lock mechanism was implemented so that only one instance of `run.py` will be run at any time. The shell script that is invoked by cron scheduler will first check if the lock file exist in the designated location. If the lock file does not exist, the shell script will create a lock file then invoke `run.py`. If the lock file does exist, the shell script will exit straightaway.

2.5 Visualisation *Work by APG(75%) and PHM(25%). Written by APG.*

Visualisation is the final step of the analysis in RAWG and is used to plot the results and compare workflows. Since the tool was created with the idea of easily testing different workflows on the same data set, this is reflected in the style of the visualisation. The main idea is to offer a quick way to compare the results of different tool, create graphs that are suitable for publication and allow minimum customisation of content plotted.

In order to achieve this, a new part of the front end was created which would be accessible only after all the workflows linked to a single session have been completed. The new page is linked to a single session.

A collapsible sidebar on the left of the visualisation page is populated with the workflows associated with a specific session instance. Selection of a workflow reveals all the associated output csv files produced by that workflow. A user can then use a checkbox form to select or de-select files to include in particular visualisations. Having selected the desired files, a user can select their desired graph from the visualization navbar across the top of the page.

In order to facilitate the user, only one type of graph is plotted at a time. The page also presents a navigation bar that allows to select which workflows will be used in the plotting, so that the user can make comparisons between different tools or select a single one at a time. Finally, in the tab div there are controls, which change accordingly to what is currently plotted, that allow customisation.

The different tabs contain an empty SVG field with a specific id, related to the type of plot. In this way, it is possible to plot and store multiple plots but only showing one at a time. It is important to notice that the choice of colours used come from a colourblind-friendly palette, allowing all the user to properly experience the feature.

The results from the analysis are stored in CSV format, in folders outside the directory of django. For this reason, when it comes to loading the files we have implemented some new functions in the view where the CSV is read and passed to the page in the JSON format.

D3 has been chosen for creating the visualisation. It is a very flexible and powerful JavaScript library that enables mapping of the data, either in CSV or JSON format, to the document. It is a simple library to use and it also supports the addition of new types of graph in the future. The visualisation is currently available for DGE and DEE analysis. At the moment, four types of plotting options are available: volcano plot, barplot, dot plot and a heatmap.

2.5.1 Volcano Plot

The volcano plot is a scatterplot that allows user to rapidly identify significant genes. On the X axis is plotted the log2 fold change, while on the Y axis the negative log of the adjusted p value. The plot present red lines, one horizontal and two vertical to determine the area of significant results. Two sliders, outside the graph, allow the user to change the p value and log2 fold change threshold updating the lines. Different workflows are plotted in different colours and non-significant genes are plotted in grey independently from the workflow which generated them. Displaying multiple results into the same graph may seem confusing, but the majority of significant differentially expressed genes will show differences between workflows. It is also possible to plot single workflow at a time to get more specific visualisation. In order to help the user navigate the plot, when hovering over a point the gene name and other information is displayed in a text label (Figure 16).

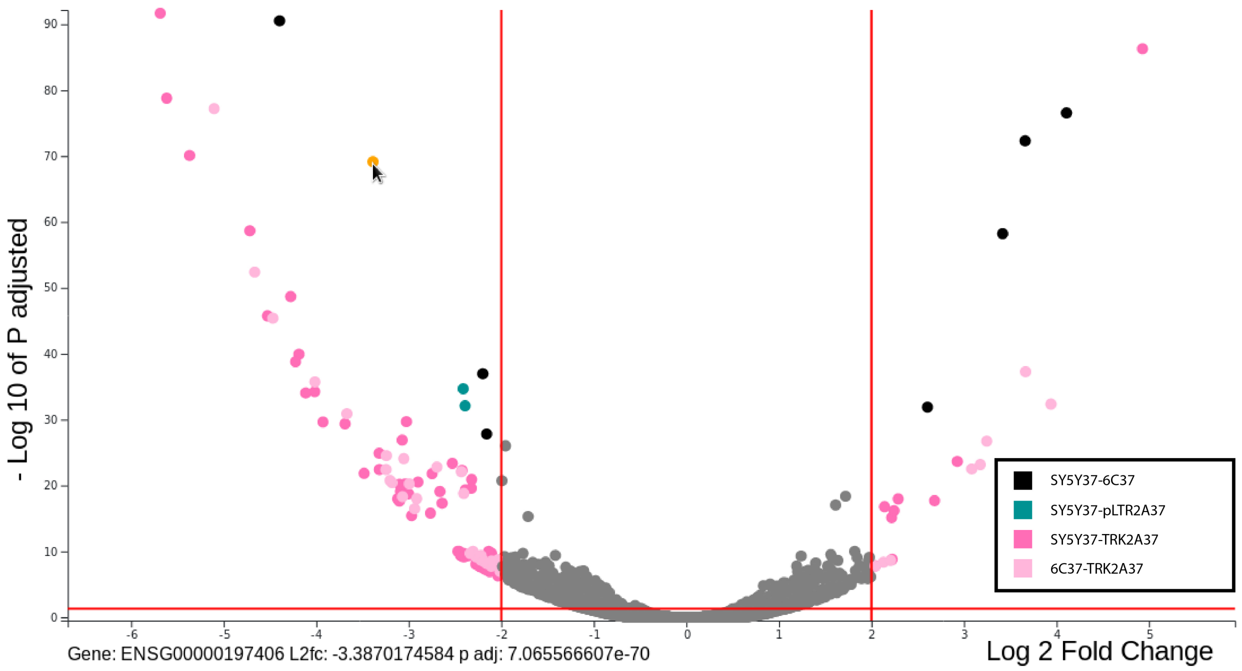


Figure 16: Graph obtained using the visualisation tools in RAWG. The volcano plot shows the log 2 fold change on the x axis and the negative log of the p adjusted on the y axis. The area of significance, selected by the user is highlighted by the red bars. The information about the gene selected is visible below the x axis.

2.5.2 Barplot

The barplot is used to show the number of significant differentially expressed genes for each workflow. The data is filtered based on thresholds set by the user and the number of significant genes is reported. This allows user to compare quantitative differences among workflows at a glance (Figure 17).

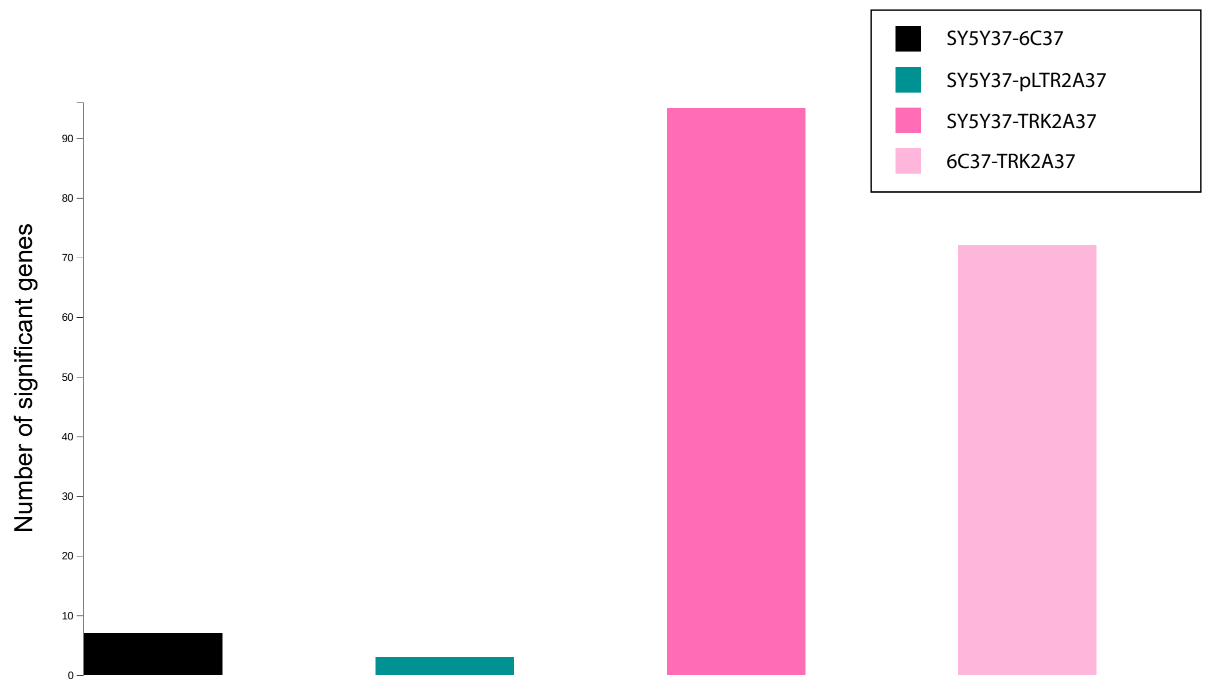


Figure 17: Graph obtained using the visualisation tools in RAWG. It shows the number of significant genes, with thresholds selected by the user for all the workflows selected.

3 Results and Dicussions

3.1 Choice of CWL Runner *Written by ZY.*

CWL's reference runner, cwltool, was the main workflow executor used in this project. Cwltool supports local execution on a single machine only and provides minimal resource management feature. This is suitable for our development purpose. For deployment in a production environment, a CWL executor that supports better resource pooling should be used, especially for hardware with high CPU core counts. Cwltool executes all the tools sequentially which means single threaded tools would drastically reduce efficiencies. Even for multithreaded tools, Amdahl's law indicates that multithreading performance does not scale well beyond certain threshold. In practice, I found that the aligners we incorporated into workflows do not scale well above 12 threads. Hence, for a workstation with 48 cores, it would be more efficient to run four aligner steps in parallel than run one aligner with 48 threads. Despite cwltool recently adding a parallel flag, it is an experimental feature and our workflows do not behave stably in my tests. CWL specification does include a section where the user can indicate resource requirements for a tool but the detailed implementation solely depends on the runner. In theory, computational resource can be cleverly distributed by the runner based on the tool's requirements. For example, on a 48 core single-node server, four aligners each using 12 threads can run simultaneously then eight single threaded programs can be run together with other tools fully utilise available resources.

3.2 Containers for Modern Bioinformatics Software Distribution *Written by ZY.*

Container is a light-weight virtualisation technology that enables running software in a specifically designed environment. By using containers, such as Docker, users are liberated from the burden of resolving certain software dependencies for a particular tool. Software development in academia is not agile, which means less maintained tools may have obsolete dependencies. Containerising old tools will ensure they still function well on modern operating systems. A set of well maintained containers would be especially valuable for the bioinformatics community to save researchers' time from technical details. Increasingly, analysis pipelines are distributed as container images where a single container encapsulate all the tools required for a certain workflow. This is a great way to standardise data analysis workflows and improve data reproducibility. However, updating a certain tool in a pipeline will mean to release a new version of the entire container. Therefore, we think

CWL's approach, using containerised tool for each step of a workflow rather than containerise the whole workflow (eg. GATK[41]), is a better way forward. We envision that workflows can be standardised using a list of container images with specified versions. In addition, separating each tool's container will allow for better task orchestration on different platforms.

3.3 Website *Written by PHM.*

The Django web framework has proven to be an extremely effective tool for building database-oriented websites. The ability to define the database tables as Python classes means that the tables could be referenced directly when building html forms. In this way, the upload and selection forms are mapped directly to the appropriate database entries. This means that the code is more readable and simpler to debug whilst enabling a robust architecture.

Django comes with templates that handle the request-response cycles. These contributed to our clean and maintainable codebase, by enabling execution of complex database operations with a few lines of code. Additionally, the rich ecosystem surrounding the Django framework meant that an abundance of packages and modules were available when I was faced with a particular challenge. For example, I used the `FileSystemStorage` module to save user uploads to a folder outside of the webportal directory. This was necessary to separate user uploaded files from our codebase. The workflow output files could then be requested on demand and copied into the webportal directory when required.

The result is that users are freed from the requirement of manually error-checking workflow errors, allowing them to focus on designing workflows for their research need.

4 Applications

4.1 Workflow Comparison *Work by ZY. Written by ZY.*

To fully demonstrate RAWG's capability of running multiple workflows in one session, I simulated RNA-Seq data and fed it to all the workflows in RAWG that generate DGE results.

4.1.1 Methodology

Polyester is a package in R that simulate RNA-Seq data based on a set of transcripts and a fold change matrix[42]. Polyester's parameters were configured to generate a 20x coverage dataset with 101 bp read length, and the FPKM value for each transcript was set to be roughly the same. In addition, a RNA fragmentation based bias was introduced via polyester. To avoid potential problems that genes can contain multiple possible transcripts, I used *Escherichia Coli*'s cDNA dataset from Ensembl. A fold change matrix was created for all the transcripts under two conditions. Twenty transcripts were randomly selected and a 4-fold change in expression was introduced to these transcripts, while all other transcripts were set to be equal (zero fold changes). Three replicates for each condition were simulated. The generated files were in FASTA format which were shuffled and converted to FASTQ format by introducing an uniform phred score (40). The code used for data simulation is included in appendix (Listing 5 and 6). Finally, the six sets of paired-end FASTQ data were uploaded to RAWG's webportal.

4.1.2 Results and Discussion

After collecting all the DGE results, the log2 fold change values for the twenty selected genes were extracted from each analysis pipeline. The data was plotted as a heat map (Figure 18). Note that for a 4-fold change in gene expression, the theoretical log2 fold change is 2. So the result should be as close to 2 as possible.

The two CuffDiff analysed results have a section of data missing. This is because CuffDiff merges overlapping transcripts and the result is not representative for single gene differential expression, thus these data points were discarded. Consistently, DESeq2 results have a higher variance and tend to underestimate the fold change while EdgeR and CuffDiff results are similar to each other and closer to the ground truth. DESeq2's high variance results indicate that the choice of mapper and assembler plays an important role in DESeq2 analysis. For EdgeR and CuffDiff, the choice of mapper

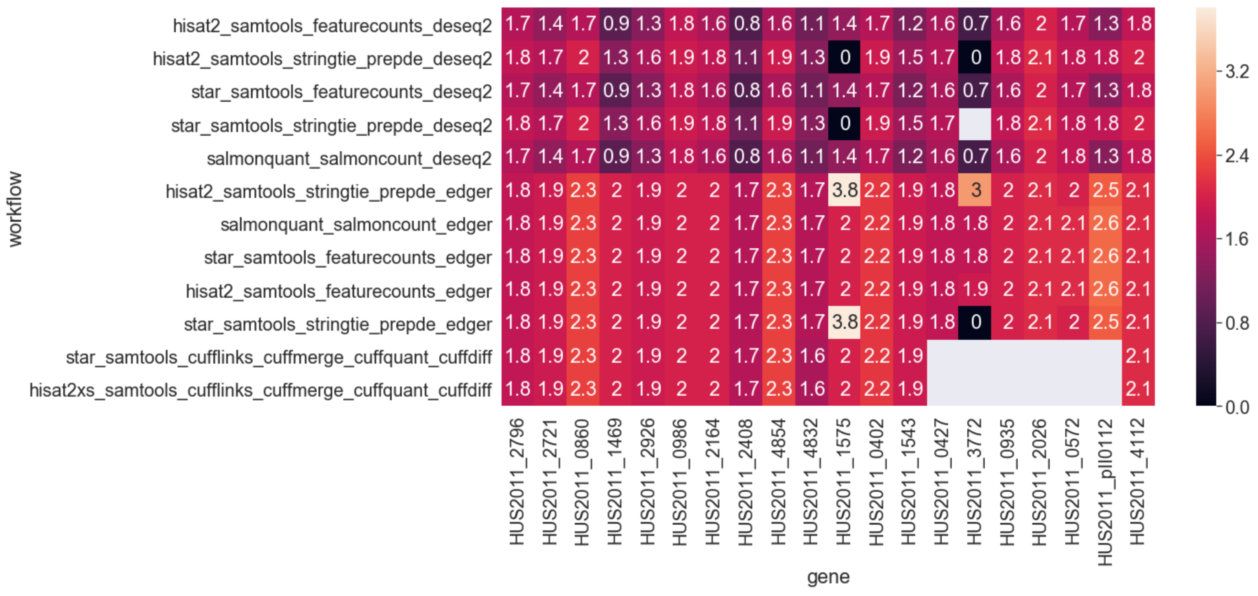


Figure 18: A heatmap of log2 fold change from DGE results with different workflows for twenty randomly chosen genes. The code for plotting the heatmap is included in appendix, Listing 7.

and assembler seems to have minimal effect which can mean that all mapper and assembler pairs can effectively map the reads to the genome and count the number of genes expressed. Interestingly, when looking at the heatmap column by column, a pattern of expression level by gene can be easily seen. This can be in part explained by gene length. For example, gene HUS2011_3772 is only 128 bp in length which would be difficult to quantify, hence the large deviation from ground truth for all the analyses.

5 Future work

5.1 Server-side *Written by ZY.*

In this project, cwltool was used which provides minimal resource management feature and only suitable for local single machine execution. We would like to investigate other CWL runners to better utilise hardware resources, in particular, Toil[13] and CWL-Airflow[15]. Toil is designed with a focus on massively parallel machines as well as public clouds. It would be interesting to see how Toil's scheduler runs a workflow in parallel compares to cwltool. CWL-Airflow not only provides a fully featured scheduler, it also integrates a resource monitoring front-end. With CWL-Airflow, the system administrators can visually inspect the resource usage of each workflow and the time it took to run each step of a workflow. In addition, CWL-metrics is a system specifically designed to collect workflow runtime metrics which also provide a graphical way of monitoring resource usage[43]. I think those ecosystem developed around CWL make it a truly wonderful tool to use.

5.2 Database *Written by PHM.*

The current database architecture used by RAWG provides a stable framework on which server-side scripts can be executed. However, there are improvements that could be incorporated into future releases. The current many to one, non-mandatory relationship between the Genome and Session table is used to define the genome files when the pre-indexed option has been selected. In these cases the server-side scripts extract the genome files from the Genome table and the duplicated fields in the Session table are not used, creating database redundancy. This could be addressed by splitting the Session table into two tables as in Figure 19.

In an optimally designed database, the Session table would contain a primary key and identifier as before, in addition to two foreign keys which link to either the Pre-Index table or the User-Index table. Selecting to use a pre-indexed genome would associate the session instance with one of our pre-indexed genomes. However, choosing a user defined index would store the uploaded genome files in a separate User-Index table. In this way the Session table is decoupled from the actual file uploads and instead acts as a pointer, directing the server-side scripts to either the Pre-Index or User-Index tables which then contain the actual file paths.

Further normalisation is possible in the Samples table. Having split conditions out into a separate table, we achieved normalisation under the second normal form. However we did not eliminate the transitive dependency in which read_2 was dependent on the library type rather than on the primary key. This could have been addressed by partitioning the read_1 and read_2 fields into a separate Read table which would be linked to Samples by a many to one relationship.

There is also an opportunity to improve the normalisation of the Workflow table. Currently we use the paths field to store a dictionary containing the file paths to the output csv files. With more time we would have liked to replace this dictionary with a foreign key to a new Paths table, eliminating the need for non atomic data types in our Workflow table.

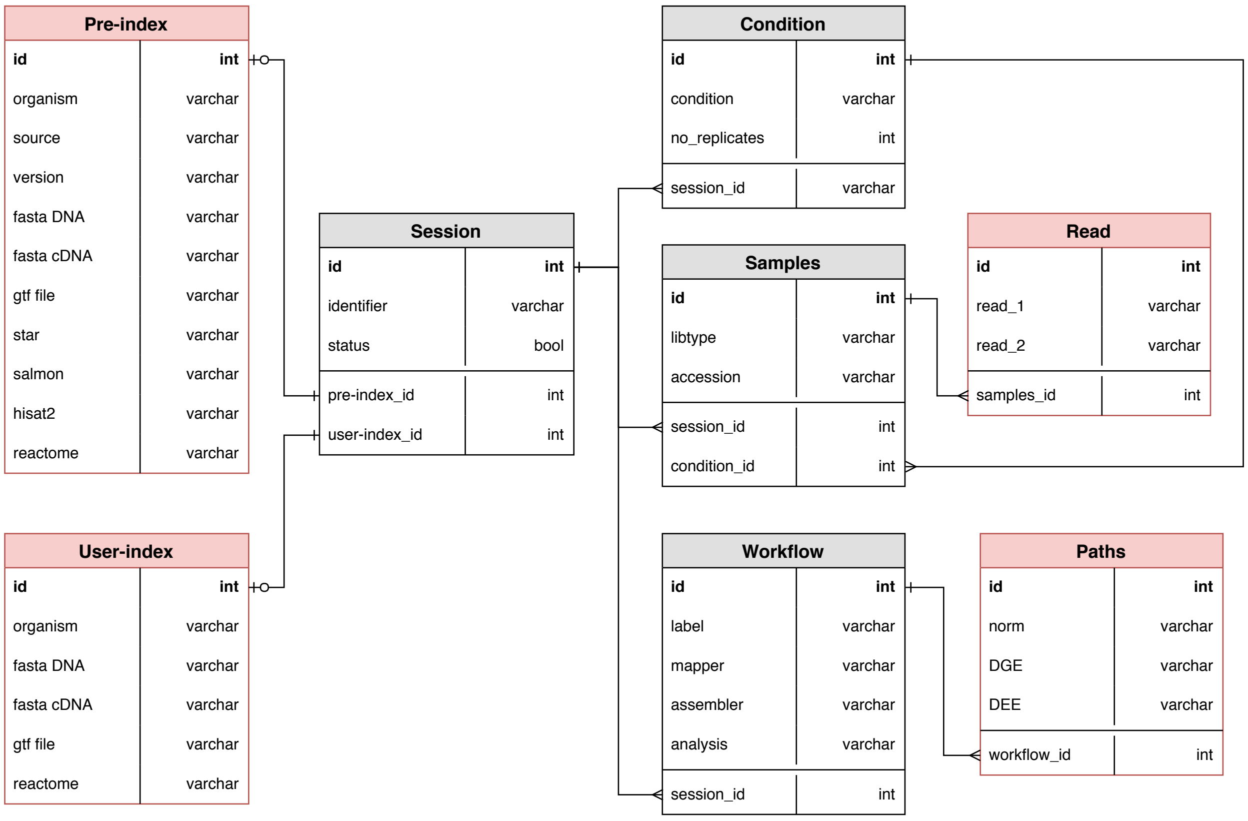


Figure 19: Idealised Database Schema. New tables are shown in red. The Queue table has been removed for clarity.

5.3 Visualisation *Written by APG.*

The visualisation capabilities included in RAWG are useful to rapidly understand the differences between different workflows. In the future, it would be possible to enhance the graphs already available and add more plots to make comparisons even easier to see and more interactive.

It would be useful to integrate a heatmap, possibly showing different workflows' results side by side. Heatmaps are largely used in DGE as they allow the user to locate basic gene patterns without further analysis. The implementation of a heatmap would further improve quantitative comparisons between workflows, which at the moment is only possible through the barplot.

For existing plots, future modifications may focus mainly on the volcano plot. The barplot allows users to understand differences in the results at a glance. The volcano plot, however, allows numerous improvements. The ability to zoom in the plot and navigate around would facilitate the analysis of genes that show smaller differences between workflows, even though the importance of these differences decreases together with it. Another interesting functionality would be expanding the "on click" action. Currently it is not used, but in the future it may be useful to expand this with a "gene card" which gives information about the gene selected and linking the plot with external resources.

5.4 CWL scripts *Written by FJ*

In this project we provided the user with a plethora of options for DGE analysis, therefore the user has 9 different DGE assembler and analysis combinations to pick from. However, DEE and DIE tools are far more sparse with only a single assembler and analysis tool available for each. In our report we have stressed the importance of different options and our workflow comparison study showcases why a user should want to fine tune their workflows. Thus, future updates of RAWG would benefit from more DIE and DEE options, for example IsoformSwitchAnalyzeR[44] is a MISO alternative (DIE analysis tool) and has compatibility with alignment free assemblers such as Salmon, or DIEGO[45] which is a pythonic alternative to DEXSeq. Along with adding more options for the analysis types we already provide, I would also want to provide more analyses. For example, single cell and fusion gene discovery are a two common RNA-Seq analyses which are already supported by the likes of SevenBridges and Galaxy.

Adhering to making the workflow generation as easy as possible, we set defaults for many parameters of tools, which greatly simplified users' setup process but also limited users' choices. Sometimes the default parameters may not be applicable to users' use case. Hence, we want to open the possibility of adjusting some of the parameters to give users more freedom.

6 Conclusions *Written by PHM.*

The rapid pace of innovation in the field of sequencing has meant an explosion in the number of tools available for analysis. This creates problems when interpreting differences of downstream analyses between different RNA-Seq pipelines because there are multiple junctures at which discrepancies can occur. This issue is compounded since there are numerous parameters within each step of the pipeline that a user can manually adjust. Failure to apply these consistently across all inter-pipeline comparisons will invalidate any further analysis, which are often computationally intensive to run. The result is that inter-pipeline comparisons of RNA-seq analysis are difficult to interpret.

In this project, we have designed RAWG to address this unmet need. RAWG is a workflow framework that can dynamically generate data analysis pipelines based on user selected tools. Users are freed from manually running multiple command-line tools with specific parameters to conduct RNA-Seq analysis. RAWG uses contemporary workflow system, Common Workflow Language, to improve data reproducibility and workflow reusability. High level logical programming was used to permit a computationally efficient inter-pipeline comparison by reusing steps that are common to two or more user defined pipelines. RAWG marries this innovative programming with an website that enables intuitive selection of the current gold standard RNA-Seq workflows without needing to download specialised software. We addressed the issue of data provenance by providing the user with a Research Object which captures the full derivative history of the end results. Software containers were provided to create a consistent software environment, enhancing data reproducibility. Finally we have designed a data visualisation suite where users can interact with their workflows to investigate the effects of varying individual pipeline components to gain a better understanding of the source of variability in outcome.

References

1. Park, S. T. & Kim, J., 2016 Trends in next-generation sequencing and a new era for whole genome sequencing. *International Neurourology Journal* **20**, 76–83. (doi:10.5213/inj.1632742.371).

2. Kuchta, K., Towpik, J., Biernacka, A., Kutner, J., Kudlicki, A., Ginalski, K. & Rowicka, M., 2018 Predicting proteome dynamics using gene expression data. *Scientific Reports* **8**, 1–13. (doi:10.1038/s41598-018-31752-4).

3. Margulies, M., Egholm, M., Altman, W. E., Attiya, S., Bader, J. S., Bemben, L. A., Berka, J., Braverman, M. S., Chen, Y.-J., Chen, Z. *et al.*, 2005 Genome Sequencing in Open Microfabricated High Density Picoliter Reactors. *Nature* **437**, 376–380.

4. Novo, S. M., Banerjee, S., Benoit, V. A., Rasolonjatovo, I. M. J., Bridgham, J. A., Golda, G. S., Bentley, D. R., Kumar, A., Mullikin, J. C., Aniebo, I. C. *et al.*, 2008 Accurate whole human genome sequencing using reversible terminator chemistry. *Nature* **456**, 53–59. (doi:10.1038/nature07517).

5. Jain, M., Olsen, H. E., Paten, B. & Akeson, M., 2016 The Oxford Nanopore MinION: Delivery of nanopore sequencing to the genomics community. *Genome Biology* **17**, 1–11. (doi:10.1186/s13059-016-1103-0l).

6. Whitney, H. M., Andrew, P., Steiner, U., Glover, B. J., Kolle, M. & Chittka, L., 2009 Floral Iridescence, Produced by Diffractive Optics, Acts As a Cue for Animal Pollinators. *Science* **323**, 133–138. (doi:10.1126/science.1166256).

7. Bittner, A., Liu, X., Zhao, S., Fung-Leung, W.-P. & Ngo, K., 2014 Comparison of RNA-Seq and Microarray in Transcriptome Profiling of Activated T Cells. *PLoS ONE* **9**, e78644. (doi:10.1371/journal.pone.0078644).

8. Leipzig, J., 2017 A review of bioinformatic pipeline frameworks. *Briefings in bioinformatics* **18**, 530–536. ISSN 1477-4054. (doi:10.1093/bib/bbw020).

9. Amstutz, P., Crusoe, M. R., Tijanić, N., Chapman, B., Chilton, J., Heuer, M., Kartashov, A., Leehr, D., Ménager, H., Nedeljkovich, M. *et al.*, 2016. Common workflow language, v1.0. (doi:10.6084/m9.figshare.3115156.v2).

10. O'Connor, B. D., Yuen, D., Chung, V., Duncan, A. G., Liu, X. K., Patricia, J., Paten, B., Stein, L. & Ferretti, V., 2017 The Dockstore: enabling modular, community-focused sharing of Docker-based genomics tools and workflows. *F1000Research* **6**, 52. ISSN 2046-1402. (doi: 10.12688/f1000research.10137.1).

11. Simmhan, Y. L., Plale, B. & Gannon, D., 2005 A survey of data provenance in e-science. *ACM SIGMOD Record* **34**, 31. ISSN 01635808. (doi:10.1145/1084805.1084812).

12. Khan, F. Z., Soiland-reyes, S., Sinnott, R. O., Lonie, A., Goble, C. & Crusoe, M. R., 2018 Sharing interoperable workflow provenance: A review of best practices and their practical application in CWLProv. *GigaScience* **Preprint**, 1–24. (doi:10.5281/ZENODO.1966881).

13. Vivian, J., Rao, A., Nothaft, F. A., Ketchum, C., Novak, A., Pfeil, J., Narkizian, J., Deran, A. D., Schmidt, H., Amstutz, P. *et al.*, 2016 Rapid and efficient analysis of 20,000 RNA-seq samples with Toil. *bioRxiv* **2**, 062497. (doi:10.1101/062497).

14. Kaushik, G., Ivkovic, S., Simonovic, J., Tijanic, N., Davis-Dusenbery, B. & Kural, D., 2017 Rabix: an Open-Source Workflow Executor Supporting Recomputability and Interoperability of Workflow Descriptions. *Pacific Symposium on Biocomputing. Pacific Symposium on Biocomputing* **22**, 154–165. ISSN 2335-6936. (doi:10.1142/9789813207813_0016).

15. Kotliar, M., Kartashov, A. & Barski, A., 2018 CWL-Airflow: a lightweight pipeline manager supporting Common Workflow Language. *bioRxiv* p. 249243. (doi:10.1101/249243).

16. Afgan, E., Baker, D., Batut, B., Van Den Beek, M., Bouvier, D., Ech, M., Chilton, J., Clements, D., Coraor, N., Grüning, B. A. *et al.*, 2018 The Galaxy platform for accessible, reproducible and collaborative biomedical analyses: 2018 update. *Nucleic Acids Research* **46**, 537–544. (doi:10.1093/nar/gky379).

17. Grennan, A. K., 2006 Genevestigator. Facilitating Web-Based Gene-Expression Analysis. *Plant Physiology* **141**, 1164–1166. (doi:10.1104/pp.104.900198).

18. Browning, J., Johnson, I. J., Spulber, I., Lee, W.-P., Stachyra, A. L., Arsenijevic, V., Suciu, M. C., Ji, S.-G., Popovic, M., Glotova, I. *et al.*, 2019 Fast and accurate genomic analyses using genome graphs. Technical report, Seven Bridges Genomics, Cambridge,. (doi:10.1038/s41588-018-0316-4).

19. da Veiga Leprevost, F., Grüning, B. A., Alves Aflitos, S., Röst, H. L., Uszkoreit, J., Barsnes, H., Vaudel, M., Moreno, P., Gatto, L., Weber, J. *et al.*, 2017 BioContainers: an open-source and

community-driven framework for software standardization. *Bioinformatics (Oxford, England)* ISSN 13674811. (doi:10.1093/bioinformatics/btx192).

20. Digan, W., Countouris, H., Barritault, M., Baudoin, D., Laurent-Puig, P., Blons, H., Burgun, A. & Rance, B., 2017 An architecture for genomics analysis in a clinical setting using Galaxy and Docker. *GigaScience* **6**, 1–9. (doi:10.1093/gigascience/gix099).

21. Amstutz, P., 2015 Common Workflow Language. Technical report.

22. Zaleski, C., Jha, S., Gingeras, T. R., Batut, P., Davis, C. A., Chaisson, M., Dobin, A., Schlesinger, F. & Drenkow, J., 2012 STAR: ultrafast universal RNA-seq aligner. *Bioinformatics* **29**, 15–21. (doi:10.1093/bioinformatics/bts635).

23. Kim, D., Langmead, B. & Salzberg, S. L., 2015 HISAT: A fast spliced aligner with low memory requirements. *Nature Methods* **12**, 357–362. (doi:10.1038/nmeth.3317).

24. Li, H., Handsaker, B., Wysoker, A., Fennell, T., Ruan, J., Homer, N., Marth, G., Abecasis, G., Durbin, R., Project, G. *et al.*, 2009 The Sequence Alignment/Map format and SAMtools. *Bioinformatics applications note* **25**, 2078–2079. (doi:10.1093/bioinformatics/btp352).

25. Pertea, M., Pertea, G. M., Antonescu, C. M., Chang, T. C., Mendell, J. T. & Salzberg, S. L., 2015 StringTie enables improved reconstruction of a transcriptome from RNA-seq reads. *Nature Biotechnology* **33**, 290–297. (doi:10.1038/nbt.3122).

26. Roberts, A., Pachter, L., Kelley, D. R., Pertea, G., Goff, L., Pimentel, H., Kim, D., Rinn, J. L., Trapnell, C. & Salzberg, S. L., 2012 Differential gene and transcript expression analysis of RNA-seq experiments with TopHat and Cufflinks. *Nature Protocols* **7**, 562–579. (doi: 10.1038/nprot.2012.016).

27. Patro, R., Duggal, G., Love, M. I., Irizarry, R. A. & Kingsford, C., 2017 Salmon provides fast and bias-aware quantification of transcript expression. *Nature Methods* **14**, 417–419. (doi: 10.1038/nmeth.4197).

28. Liao, Y., Smyth, G. K. & Shi, W., 2014 FeatureCounts: An efficient general purpose program for assigning sequence reads to genomic features. *Bioinformatics* **30**, 923–930. (doi:10.1093/ bioinformatics/btt656).

29. Katz, Y., Wang, E. T., Airoldi, E. M. & Burge, C. B., 2010 Analysis and design of RNA sequencing experiments for identifying isoform regulation. *Nature Methods* **7**, 1009–1015. (doi: 10.1038/nmeth.1528).

30. Anders, S., Pyl, P. T. & Huber, W., 2015 HTSeq-A Python framework to work with high-throughput sequencing data. *Bioinformatics* **31**, 166–169. (doi:10.1093/bioinformatics/btu638).

31. Love, M. I., Huber, W. & Anders, S., 2014 Moderated estimation of fold change and dispersion for RNA-seq data with DESeq2. *Genome Biology* **15**, 1–21. (doi:10.1186/s13059-014-0550-8).

32. Pachter, L., Goff, L., Trapnell, C., Rinn, J. L., Sauvageau, M. & Hendrickson, D. G., 2012 Differential analysis of gene regulation at transcript resolution with RNA-seq. *Nature Biotechnology* **31**, 1–19. (doi:10.1038/nbt.2450).

33. Leek, J. T., Jaffe, A. E., Langmead, B., Salzberg, S. L., Pertea, G. & Frazee, A. C., 2015 Ballgown bridges the gap between transcriptome assembly and expression analysis. *Nature Biotechnology* **33**, 243–246. (doi:10.1038/nbt.3172).

34. Edgren, R. A. & Stanczyk, F. Z., 1999 Nomenclature of the gonane progestins. *Contraception* **60**, 313. ISSN 00107824. (doi:10.1016/S0010-7824(99)00101-8).

35. Anders, S., Reyes, A. & Huber, W., 2012 Detecting differential usage of exons from RNA-seq data. *Genome Research* **22**, 2008–2017. (doi:10.1101/gr.133744.111).

36. Benjamini, Y. & Hochberg, Y., 2018 Controlling the False Discovery Rate: A Practical and Powerful Approach to Multiple Testing. *Journal of the Royal Statistical Society: Series B (Methodological)* **57**, 289–300. (doi:10.1111/j.2517-6161.1995.tb02031.x).

37. Sergushichev, A. A. An algorithm for fast preranked gene set enrichment analysis using cumulative statistic calculation (doi:10.1101/060012).

38. Wang, K., Li, M. & Bucan, M., 2007 Pathway-Based Approaches for Analysis of Genomewide Association Studies. *The American Journal of Human Genetics* **81**, 1278–1283. (doi:10.1086/522374).

39. Subramanian, A., Tamayo, P., Mootha, V. K., Mukherjee, S., Ebert, B. L., Gillette, M. A., Paulovich, A., Pomeroy, S. L., Golub, T. R., Lander, E. S. *et al.*, 2005 Gene set enrichment analysis: A knowledge-based approach for interpreting genome-wide expression profiles. *PNAS* **102**, 15545–15550.

40. Robinson, M., Soiland-Reyes, S., Crusoe, M. R., Goble, C., Robinson, M., Soiland-Reyes, S., Crusoe, M. R. & Goble, C., 2017 CWL Viewer: the common workflow language viewer. *F1000Research* **6**. (doi:10.7490/F1000RESEARCH.1114375.1).

41. Depristo, M. A., Banks, E., Poplin, R., Garimella, K. V., Maguire, J. R., Hartl, C., Philippakis, A. A., Angel, G. D., Rivas, M. A., Hanna, M. *et al.*, 2011 A framework for variation discovery and genotyping using next-generation dna sequencing data. *Nature Genetics* **43**, 491–498. (doi:10.1038/ng.806).

42. Frazee, A. C., Jaffe, A. E., Langmead, B. & Leek, J. T., 2015 Polyester: Simulating RNA-seq datasets with differential transcript expression. *Bioinformatics* **31**, 2778–2784. ISSN 14602059. (doi:10.1093/bioinformatics/btv272).

43. Ohta, T., Tanjo, T. & Ogasawara, O., 2018 Accumulating computational resource usage of genomic data analysis workflow to optimize cloud computing instance selection. *bioRxiv* p. 456756. (doi:10.1101/456756).

44. Vitting-Seerup, K. & Sandelin, A., 2017 The Landscape of Isoform Switches in Human Cancers. *Molecular Cancer Research* **15**, 1206–1220. (doi:10.1158/1541-7786.mcr-16-0459).

45. Doose, G., Bernhart, S. H., Wagener, R. & Hoffmann, S., 2018 DIEGO: Detection of differential alternative splicing using Aitchison's geometry. *Bioinformatics* **34**, 1066–1068. (doi:10.1093/bioinformatics/btx690).

Appendix

```python
def featurecounts(self):
    # inputs
    self.cwl_workflow["inputs"]["featurecounts_script"] = "File"
    self.cwl_input["featurecounts_script"] = {
        "class": "File",
        "path": f"{self.root}/RNASeq/scripts/featurecount.R"
    }
    # outputs
    self.cwl_workflow["outputs"][f"{self.name}_out"] = {
        "type": "Directory",
        "outputSource": f"{self.name}_folder/out"
    }
    # steps
    self.cwl_workflow["steps"][self.name] = {
        "run": f"{self.root}/RNASeq/cwl-tools/docker/featurecounts.cwl",
        "in": {
            "input_script": "featurecounts_script",
            "bam_files": [f"{self.previous_name}_{i+1}/samtools_out" \
                            for i in range(self.num)],
            "gtf": "annotation",
            "threads": "threads",
            "metadata": "metadata"
        },
        "out": ["gene_count_output"]
    }

    # foldering
    self.cwl_workflow["steps"][f"{self.name}_folder"] = {
        "run": f"{self.root}/RNASeq/cwl-tools/folder.cwl",
        "in": {
            "item": f"{self.name}/gene_count_output",
            "name": {"valueFrom": self.name}
        },
        "out": ["out"]
    }

    # graph
    self.graph.add_node(pydot.Node(self.name, label="Featurecounts"))
    self.add_edge()
    self.add_annotation()
    self.add_metadata()
```

Listing 4: A method of cwl_writer class that generate part of workflow for one step in the analysis workflow.

```r
library(polyester)
library(Biostrings)
fastapath = "./GenomeIndex/Escherichia_Coli/Escherichia_coli.HUSEC2011CHR1.cdna.all.fa"
numtx = count_transcripts(fastapath)
fold_changes = matrix(data=1, nrow=numtx, ncol=2)
for (pos in c(3030,2260,806,2955,661,1094,1703,4006,3160,1220)){
    fold_changes[pos, 1] = 4
}
for (pos in c(2398,2642,5088,1169,5066,1809,636,1777,112,4346)){
    fold_changes[pos, 2] = 4
}
fasta = readDNAStringSet(fastapath)
readspertx = round(20 * width(fasta) / 100)
simulate_experiment(fastapath, reads_per_transcript=readspertx, fold_changes=fold_changes,
num_reps = c(3, 3), outdir='simulated_reads', seed=1, readlen=100,
distr="empirical", error_model="illumina5", bias="rnaf")
```

Listing 5: R code using Polyester package to simulate RNA-Seq reads.

```python
from Bio import SeqIO
import random

def shuffle(seq1, seq2):
        seq_1 = list(SeqIO.parse(seq1, "fasta"))
        seq_2 = list(SeqIO.parse(seq2, "fasta"))
        ind = list(range(len(seq_1)))
        random.shuffle(ind)
        SeqIO.write([seq_1[i] for i in ind], f"./shuffled/{seq1}", "fasta")
        SeqIO.write([seq_2[i] for i in ind], f"./shuffled/{seq2}", "fasta")


for i in range(1, 7):
        shuffle(f"sample_0{i}_1.fasta", f"sample_0{i}_2.fasta")
```

Listing 6: Python code for shuffling RNA-Seq reads.

```python
import pandas as pd
import os
from BCBio import GFF
import seaborn as sns

conv = {}
with open("../../Escherichia_coli.HUSEC2011CHR1.42.gtf") as f:
    for i in range(5):
        f.readline()
    for line in f.readlines():
        tmp = {}
        for i in line.strip().split("\t")[-1].split(";")[:-1]:
            s = i.strip().split(" ")
            tmp[s[0]] = s[1][1:-1]
        if "gene_id" in list(tmp) and "gene_name" in list(tmp):
            conv[tmp["gene_name"]] = tmp["gene_id"]

results = {}
for file in os.listdir("./"):
    name = file.split(".")
    if name[1] == "csv":
        results[name[0]] = pd.read_csv(file, index_col=0)\
                                .rename(index=conv).round(1)

log2change = []
for data in results:
    tmp = results[data].loc[["HUS2011_2796",
                            "HUS2011_2721",
                            "HUS2011_0860",
                            "HUS2011_1469",
                            "HUS2011_2926",
                            "HUS2011_0986",
                            "HUS2011_2164",
                            "HUS2011_2408",
                            "HUS2011_4854",
                            "HUS2011_4832",
                            "HUS2011_1575",
                            "HUS2011_0402",
                            "HUS2011_1543",
                            "HUS2011_0427",
                            "HUS2011_3772",
                            "HUS2011_0935",
                            "HUS2011_2026",
                            "HUS2011_0572",
                            "HUS2011_pII0112",
                            "HUS2011_4112"], "log2foldchange"]
    tmp.name = data
    log2change.append(tmp)

change = abs(pd.concat(log2change, axis=1))

```

```
cols = ['hisat2_samtools_featurecounts_deseq2',
    'hisat2_samtools_stringtie_prepde_deseq2',
    'star_samtools_featurecounts_deseq2',
    'star_samtools_stringtie_prepde_deseq2',
    'salmonquant_salmoncount_deseq2',
    'hisat2_samtools_stringtie_prepde_edger',
    'salmonquant_salmoncount_edger',
    'star_samtools_featurecounts_edger',
    'hisat2_samtools_featurecounts_edger',
    'star_samtools_stringtie_prepde_edger',
    'star_samtools_cufflinks_cuffmerge_cuffquant_cuffdiff',
    'hisat2xs_samtools_cufflinks_cuffmerge_cuffquant_cuffdiff'
]

change = abs(change)

change = change[cols].transpose()
change.index.names = ["workflow"]
change.columns.names = ["gene"]
sns.set(rc={'figure.figsize':(16,8)}, font_scale=1.8)
sns.heatmap(change, annot=True)
```

Listing 7: Python code for plotting the heatmap.

List of CWL script tools inputs and outputs

ballgown

Inputs:

1. input_script: R ballgown script.
2. tablemaker_output: directory of the tablemaker output
3. metadata: metadata csv file.
4. condition: string giving the condition of interest in the metadata columns

Outputs:

1. DGE_res.csv
2. DTE_res.csv

cuffdiff

Inputs:

1. output: name for output directory
2. threads: number of cores to use
3. label: the factors of the condition of interest
4. FDR: the FDR rate to which to label as significant, could default to 0.05.
5. merged_gtf: merged gtf annotation file in gtf format
6. condition1_files: cxb files of all samples for condition 1. cuffquant output
7. condition2_files: cxb files of all samples for condition 2. cuffquant output

Outputs:

1. output: names after inputs.output. directory containing all cuffdiff output files

cufflinks

Inputs:

1. gtf: annotation file in gtf format
2. output: name for output directory
3. threads: number of threads to use.
4. bam: bam file

Outputs:

1. output: named after output_dir. directory with all cufflink file outputs
2. transcripts.gtf

cuffmerge:

Inputs:

1. output: string stating the output directory name
2. gtf: annotation file in gtf format
3. threads: number of threads to use.
4. fasta: fasta file used in indexing
5. cufflinks_output: transcripts.gtf files generated by cufflinks

Outputs:

1. output: output directory named after input.output
2. merged.gtf

cuffnorm:

Inputs:

1. output: name for output directory
2. threads: number of threads to use.
3. merged_gtf: merged gtf annotation file in gtf format
4. condition1_files: cxb files of all samples for condition 1. cuffquant output
5. condition2_files: cxb files of all samples for condition 2. cuffquant output

Outputs:

1. output: named after inputs.output. file contained normalised gene count matrix

cuffquant:

Inputs:

1. output: name for output directory
2. threads: number of threads to use.
3. merged_gtf: merged gtf annotation file in gtf format
4. bam: bam file

Outputs:

1. output: named after inputs.output. Directory containing all quant files
2. abundances.cxb

DESeq2:

Inputs:

1. input_script: R DESeq2 script.
2. count_matrix: gene count matrix file
3. metadata: metadata csv file.

Outputs:

1. DGE_results.csv

DEXSeq:

Inputs:

1. input_script: R DEXSeq script
2. count_matrix: exon count matrix file
3. gff: directory containing gff file from htseq prepare
4. metadata: metadata csv file.
5. threads: number of threads to use

Outputs:

1. DEE_results.csv

edger:

Inputs:

1. input_script: R edger script
2. condition: string giving the condition of interest in the metadata columns
3. count_matrix: gene counts matrix
4. metadata: metadata csv file.

Outputs:

1. DGE_res.csv

featurecounts:

Inputs:

1. input_script: R featurecounts script
2. bam_files: all bam files
3. gtf: annotation in ftd format
4. threads: number of threads to use.
5. metadata: used to see the libType of each sample.

Outputs:

1. gene_count_matrix.csv

fgsea:

Inputs:

1. input_script: R fgsea input_script
2. de_results: DGE res file
3. gene_set: file containing gene set information in long form.

ouputs:

1. gsea_res.csv

hisat2_align:

Inputs:

1. threads: number of threads
2. index_directory: path to hisat2 index
3. first_pair: first fastq file
4. second_pair: second fastq file
5. output: output name
6. XSTag: Tag to use ("–dta" or "–dta-cufflinks")

hisat2_build:

Inputs:

1. fasta: fasta file
2. threads: number of threads to use.
3. output: name to use for output

Outputs:

1. output: samfile with a basename given from inputs.output
2. log

htseq_count:

Inputs:

1. input_script: python htseq_count script
2. pairedend: logical
3. stranded: logical
4. input_format: bam or sam
5. sorted_by: pos
6. gff: gff file from htseq_annotation
7. bam: bam or sam file
8. outname: name for outfile

Outputs:

1. outname: named after input.ouput

htseq_prepare:

Inputs:

1. input_script: python input script
2. gtf: gtf annotation file
3. gff_name: name for gff output

Outputs:

1. gff_name: gff file named after inputs.gff_name

hypergeo:

Inputs:

1. input_script: R HyperGeo_Script script
2. de_res: DGE res file
3. gene_set: file containing gene set information in long form.

Outputs:

1. hypergeo_res.csv

miso_index:

Inputs:

1. gtf: gtf file
2. output: output name for output directory

Outputs:

1. index_dir: output directory named after inputs.index_dir

miso_run:

Inputs:

1. threads: number of threads to use
2. lib_type: pairedend or not
3. index_directory: Directory from miso_index output
4. bam: bam file
5. read_len: read length
6. gtf: gtf file
7. min_exon_size: minimum exon size to use
8. output: output name

Outputs:

1. out_dir: directory named after inputs.out_dir

prepDE:

Inputs:

1. input_script: python prepDE script
2. stringtie_out: gtf files from stringtie

Outputs:

1. gene_count_matrix.csv
2. transcript_count_matrix.csv

salmon_count:

inputs:

1. input_script: R count script
2. gtf: annotation file in gtf format
3. metadata: metadata.csv
4. quant_results: directory with subdirectorys of outputs from salmon_quant

Outputs:

1. gene_abundance_matrix.csv
2. gene_count_matrix.csv * this one used for DGE analysis e.g. DESeq2
3. gene_length_matrix.csv

salmon_index:

Inputs:

1. fasta: fasta files
2. index_type: 2 different type of running ”fmd” or ”quasi”
3. threads: number of threads to use
4. output: directory output name

Outputs:

1. index_name: output directory with name inputs.index_name

salmon_quant:

Inputs:

1. index_directory: directory of salmon index
2. threads: number of threads to use
3. output: output name
4. first_end_fastq: if paired end. first pair fastq file
5. second_end_fastq if paired end. second pair fastq file
6. single_fastq: if single end. single fastq file

Outputs:

1. out_dir: Directory with name of inputs.out_dir

samtools:

Inputs:

1. action: samtools command (default in sort)
2. sortby: how to sort the bam file
3. threads: how many threads to use (total - 1), it is additional
4. samfile: sam file
5. outfilename: output file name

Outputs:

1. outfilename: bam file with name inputs.outfilename

STAR_index:

Inputs:

1. threads: number of threads
2. Mode: how to run STAR (use genomeGenerate)
3. output: directory name to use
4. fasta: fasta files
5. gtf: gtf file

STAR_readmap

Inputs:

1. threads: number of threads to use
2. Mode: how to run STAR (use alignReads)
3. genomeDir: directory name to use
4. sjdbGTFfile: gtf file
5. sjdbOverhang
6. readFilesIn: fastq files
7. outSAMtype: state the output wanted use("BAM SortedByCoordinate")
8. outFileNamePrefix: output name for directory

Outputs:

1. outFileNamePrefix: output directory name with name inputs.outFileNamePrefix
2. sam: sam file

stringtie

Inputs:

1. bam: bam file
2. threads: number of threads to use.
3. gtf: gtf file
4. output: output name for file

Outputs:

1. outfilename: output file name with name inputs.outfilename

tablemaker:

Inputs:

1. threads: number of threads to use.
2. merged_gtf: merged gtf from cuffmerge
3. bam: bam files to use
4. output: output name for directory.

Outputs:

1. output: Directory with name inputs.outputs