

---

# **Toil Documentation**

***Release 3.2.2a1***

**UCSC Computational Genomics Lab**

June 28, 2016



<b>1</b>	<b>Installation</b>	<b>3</b>
1.1	Prerequisites . . . . .	3
1.2	Basic installation . . . . .	3
1.3	Building & testing . . . . .	4
1.3.1	Running Mesos tests . . . . .	5
<b>2</b>	<b>Cloud installation</b>	<b>7</b>
2.1	Installation on AWS for distributed computing . . . . .	7
2.1.1	CGCloud in a nutshell . . . . .	7
2.2	Installation on Azure . . . . .	8
2.3	Installation on OpenStack . . . . .	11
2.4	Installation on Google Compute Engine . . . . .	11
<b>3</b>	<b>Running a workflow</b>	<b>13</b>
3.1	Running quick start . . . . .	13
3.2	Running CWL workflows . . . . .	13
3.3	Running a Toil pipeline in detail . . . . .	14
3.3.1	Changing the log statements . . . . .	17
3.3.2	Restarting after introducing a bug . . . . .	17
3.3.3	Getting stats from our pipeline run . . . . .	18
<b>4</b>	<b>Running in the cloud</b>	<b>19</b>
4.1	Running on AWS . . . . .	19
4.2	Running on Azure . . . . .	19
4.3	Running on Open Stack . . . . .	21
4.4	Running on Google Compute Engine . . . . .	21
<b>5</b>	<b>Command line interface and arguments</b>	<b>23</b>
5.1	Logging . . . . .	23
5.2	Stats . . . . .	23
5.3	Restart . . . . .	23
5.4	Clean . . . . .	24
5.5	Batch system . . . . .	24
5.6	Default cores, disk, and memory . . . . .	24
5.7	Job store . . . . .	24
5.8	Miscellaneous . . . . .	24
<b>6</b>	<b>Developing a workflow</b>	<b>25</b>
6.1	Scripting quick start . . . . .	25

6.2	Job basics . . . . .	25
6.3	Invoking a workflow . . . . .	26
6.4	Specifying arguments via the command line . . . . .	27
6.5	Resuming a workflow . . . . .	27
6.6	Functions and job functions . . . . .	28
6.7	Workflows with multiple jobs . . . . .	28
6.8	Dynamic job creation . . . . .	30
6.9	Promises . . . . .	30
6.10	Managing files within a workflow . . . . .	31
6.10.1	Staging of files into the job store . . . . .	33
6.11	Services . . . . .	34
6.12	Checkpoints . . . . .	35
6.13	Encapsulation . . . . .	35
<b>7</b>	<b>Toil API</b>	<b>37</b>
7.1	Job methods . . . . .	37
7.2	Job.FileStore . . . . .	41
7.3	Job.Runner . . . . .	43
7.4	Toil . . . . .	43
7.5	Job.Service . . . . .	45
7.6	FunctionWrappingJob . . . . .	45
7.7	JobFunctionWrappingJob . . . . .	46
7.8	EncapsulatedJob . . . . .	46
7.9	Promise . . . . .	46
7.10	Exceptions . . . . .	47
<b>8</b>	<b>Toil architecture</b>	<b>49</b>
8.1	Optimizations . . . . .	50
8.1.1	Read-only leader . . . . .	50
8.1.2	Job chaining . . . . .	50
8.1.3	Preemptable node support . . . . .	51
8.1.4	Caching . . . . .	51
<b>9</b>	<b>The batch system interface</b>	<b>53</b>
<b>10</b>	<b>The job store interface</b>	<b>55</b>
<b>11</b>	<b>src</b>	<b>61</b>
11.1	toil package . . . . .	61
11.1.1	Subpackages . . . . .	61
11.1.2	Submodules . . . . .	75
11.1.3	toil.common module . . . . .	75
11.1.4	toil.job module . . . . .	76
11.1.5	toil.jobWrapper module . . . . .	76
11.1.6	toil.leader module . . . . .	76
11.1.7	toil.realtimeLogger module . . . . .	76
11.1.8	toil.resource module . . . . .	77
11.1.9	toil.toilState module . . . . .	77
11.1.10	toil.version module . . . . .	77
11.1.11	toil.worker module . . . . .	77
11.1.12	Module contents . . . . .	77
<b>12</b>	<b>Indices and tables</b>	<b>79</b>
	<b>Python Module Index</b>	<b>81</b>

Toil is a workflow engine entirely written in Python. It features:

- Easy installation, e.g. `pip install toil`.
- [Common Workflow Language \(CWL\)](#) support

Complete support for the draft-3 CWL specification, allowing it to execute CWL workflows.

- [Workflow Description Language \(WDL\)](#) support

Draft support for the WDL specification, allowing it to execute WDL workflows.

- Cross platform support

Develop and test on your laptop then deploy on any of the following:

- Commercial clouds: - [Amazon Web Services](#) (including the spot market) - [Microsoft Azure](#) - [Google Compute Engine](#)
- Private clouds: - [OpenStack](#)
- High Performance Computing Environments: - [GridEngine](#) - [Apache Mesos](#) - [Parasol](#) - Individual multi-core machines

- A small API

Easily mastered, the Python user API for defining and running workflows is built upon one core class.

- Complete file and stream management:

Temporary and persistent file management that abstracts the details of the underlying file system, providing a uniform interface regardless of environment. Supports both atomic file transfer and streaming interfaces, and provides encryption of user data.

- Scalability:

Toil can easily handle workflows concurrently using hundreds of nodes and thousands of cores.

- Robustness:

Toil workflows support arbitrary worker and leader failure, with strong check-pointing that always allows re-summption.

- Efficiency:

Caching, fine grained, per task, resource requirement specifications, and support for the AWS spot market mean workflows can be executed with little waste.

- Declarative and dynamic workflow creation:

Workflows can be declared statically, but new jobs can be added dynamically during execution within any existing job, allowing arbitrarily complex workflow graphs with millions of jobs within them.

- Support for databases and services:

For example, Apache Spark clusters can be created quickly and easily integrated within a toil workflow as a service, with precisely defined time start and end times that fits with the flow of other jobs in the workflow.

- Open Source: An Apache license allows unrestricted use, incorporation and modification.

Contents:



---

## Installation

---

### 1.1 Prerequisites

- Python 2.7.x
- `pip > 7.x`

### 1.2 Basic installation

To setup a basic Toil installation use

```
pip install toil
```

Toil uses `setuptools`' `extras` mechanism for dependencies of optional features like support for Mesos or AWS. To install Toil with all bells and whistles use

```
pip install toil[aws,mesos,azure,google,encryption,cwl]
```

Here's what each extra provides:

- The `aws` extra provides support for storing workflow state in Amazon AWS. This extra has no native dependencies.
- The `google` extra is experimental and stores workflow state in Google Cloud Storage. This extra has no native dependencies.
- The `azure` extra stores workflow state in Microsoft Azure Storage. This extra has no native dependencies.
- The `mesos` extra provides support for running Toil on an [Apache Mesos](#) cluster. Note that running Toil on SGE (GridEngine), Parasol or a single machine does not require an extra. The `mesos` extra requires the following native dependencies:
  - *Apache Mesos*
  - *Python headers and static libraries*
- The `encryption` extra provides client-side encryption for files stored in the Azure and AWS job stores. This extra requires the following native dependencies:
  - *Python headers and static libraries*
  - *Libffi headers and library*
- The `cwl` extra provides support for running workflows written using the [Common Workflow Language](#).

### Apache Mesos

Only needed for the `mesos` extra. Toil has been tested with version 0.25.0. Mesos can be installed on Linux by following the instructions on <https://open.mesosphere.com/getting-started/install/>. The [Homebrew](#) package manager has a formula for Mesos such that running `brew install mesos` is probably the easiest way to install Mesos on OS X. This assumes, of course, that you already have [Xcode](#) and [Homebrew](#).

Please note that even though Toil depends on the Python bindings for Mesos, it does not explicitly declare that dependency and they will **not** be installed automatically when you run `pip install toil[mesos]`. You need to install the bindings manually. The [Homebrew](#) formula for OS X installs them by default. On Ubuntu you will need to download the appropriate `.egg` from <https://open.mesosphere.com/downloads/mesos/> and install it using `easy_install -a <path_to_egg>`. Note that on Ubuntu Trusty you may need to upgrade `protobuf` via `pip install --upgrade protobuf` **before** running the above `easy_install` command.

If you intend to install Toil with the `mesos` extra into a virtualenv, be sure to create that virtualenv with

```
virtualenv --system-site-packages
```

Otherwise, Toil will not be able to import the `mesos.native` module.

### Python headers and static libraries

Only needed for the `mesos` and `encryption` extras. The Python headers and static libraries can be installed on Ubuntu/Debian by running `sudo apt-get install build-essential python-dev` and accordingly on other Linux distributions. On Mac OS X, these headers and libraries are installed when you install the [Xcode](#) command line tools by running `xcode-select --install`, assuming, again, that you have [Xcode](#) installed.

### Libffi headers and library

[Libffi](#) is only needed for the `encryption` extra. To install [Libffi](#) on Ubuntu, run `sudo apt-get install libffi-dev`. On Mac OS X, run `brew install libffi`. This assumes, of course, that you have [Xcode](#) and [Homebrew](#) installed.

## 1.3 Building & testing

For developers and people interested in building the project from source the following explains how to setup virtualenv to create an environment to use Toil in.

After cloning the source and `cd`-ing into the project root, create a virtualenv and activate it:

```
virtualenv venv
. venv/bin/activate
```

Simply running

```
make
```

from the project root will print a description of the available Makefile targets.

Once you created and activated the virtualenv, the first step is to install the build requirements. These are additional packages that Toil needs to be tested and built, but not run:



```
make prepare
```

Once the virtualenv has been prepared with the build requirements, running

```
make develop
```

will create an editable installation of Toil and its runtime requirements in the current virtualenv. The installation is called *editable* (also known as a *development mode* installation) because changes to the Toil source code immediately affect the virtualenv. Optionally, set the `extras` variable to ensure that `make develop` installs support for optional extras. Consult `setup.py` for the list of supported extras. To install Toil in development mode with all extras run

```
make develop extras=[aws,mesos,azure,google,encryption,cwl]
```

Note that some extras have native dependencies as listed in *Basic installation*. Be sure to install them before running the above command. If you get

```
ImportError: No module named mesos.native
```

make sure you install Mesos and the Mesos egg as described in *Apache Mesos* and be sure to create the virtualenv with `--system-site-packages`.

To build the docs, run `make develop` with all extras followed by

```
make docs
```

To invoke the tests (unit and integration) use

```
make test
```

Run an individual test with

```
make test tests=src/toil/test/sort/sortTest.py::SortTest::testSort
```

The default value for `tests` is `"src"` which includes all tests in the `src` subdirectory of the project root. Tests that require a particular feature will be skipped implicitly. If you want to explicitly skip tests that depend on a currently installed *feature*, use

```
make test tests="-m 'not azure' src"
```

This will run only the tests that don't depend on the `azure` extra, even if that extra is currently installed. Note the distinction between the terms *feature* and *extra*. Every extra is a feature but there are features that are not extras, the `gridengine` and `parasol` features fall into that category. So in order to skip tests involving both the Parasol feature and the Azure extra, the following can be used:

```
make test tests="-m 'not azure and not parasol' src"
```

### 1.3.1 Running Mesos tests

See *Apache Mesos*. Be sure to create the virtualenv with `--system-site-packages` to include the Mesos Python bindings. Verify by activating the virtualenv and running `.. pip list | grep mesos`. On OS X, this may come up empty. To fix it, run the following:

```
for i in /usr/local/lib/python2.7/site-packages/*mesos*; do ln -snf $i venv/lib/python2.7/site-packages
```



---

## Cloud installation

---

### 2.1 Installation on AWS for distributed computing

We use [CGCloud](#) to provision instances and clusters in AWS. Thorough documentation of [CGCloud](#) can be found in the [CGCloud-core](#) and [CGCloud-toil](#) documentation. Brief steps will be provided to those interested in using [CGCloud](#) for provisioning.

#### 2.1.1 CGCloud in a nutshell

Setting up clusters with [CGCloud](#) has the benefit of coming pre-packaged with Toil and Mesos, our preferred batch system for running on AWS. If you encounter any issues following these steps, check official documentation which contains Troubleshooting sections.

1. `virtualenv ~/cgcloud`
2. `source ~/cgcloud/bin/activate`
3. `pip install cgcloud-core`
4. `pip install cgcloud-toil`
5. **Add the following to your `~/.profile`, use the appropriate region for your account.**
  - 5a. `export CGCLOUD_ZONE=us-west-2a`
  - 5b. `export CGCLOUD_PLUGINS="cgcloud.toil:$CGCLOUD_PLUGINS"`
6. Setup credentials for your AWS account in `~/.aws/credentials`:

```
[default]
aws_access_key_id=PASTE_YOUR_FOO_ACCESS_KEY_ID_HERE
aws_secret_access_key=PASTE_YOUR_FOO_SECRET_KEY_ID_HERE
region=us-west-2
```

7. **Register your SSH key. You can create one with `ssh-keygen`.**
  - 7a. `cgcloud register-key ~/.ssh/id_rsa.pub`
8. **Create a template *toil-box* which will contain necessary prerequisites**
  - 8a. `cgcloud create -IT toil-box`
9. **Create a small leader/worker cluster**
  - 9a. `cgcloud create-cluster toil -s 2 -t m3.large`
10. SSH into the leader: `cgcloud ssh toil-leader`

At this point, any toil script can be run on the distributed AWS cluster following instructions in [Running on AWS](#).

## 2.2 Installation on Azure

While CGCloud does not currently support cloud providers other than Amazon, Toil comes with a cluster template to facilitate easy deployment of clusters running Toil on Microsoft Azure. The template allows these clusters to be created and managed through the Azure portal.

Detailed information about the template is available [here](#).

To use the template to set up a Toil Mesos cluster on Azure, follow these steps.

1. Make sure you have an SSH RSA public key, usually stored in `~/.ssh/id_rsa.pub`. If not, you can use `ssh-keygen -t rsa` to create one.
2. Click on the deploy button above, or navigate to `https://portal.azure.com/#create/Microsoft.Template/uri` in your browser.
3. If necessary, sign into the Microsoft account that you use for Azure.
4. You should be presented with a screen resembling the following:

The screenshot shows the Microsoft Azure portal interface for creating a custom deployment. The left sidebar contains navigation options like 'New', 'Resource groups', 'All resources', 'Recent', 'App Services', 'SQL databases', 'Virtual machines (classic)', 'Virtual machines', 'Cloud services (classic)', 'Subscriptions', 'Storage accounts', 'Public IP addresses', 'Availability sets', 'Network security groups', and 'Load balancers'. The main area is divided into two tabs: 'Custom deployment' and 'Parameters'. The 'Custom deployment' tab has sections for 'Template' (Edit template), 'Parameters' (Edit parameters), 'Subscription' (cgl-hgvm), 'Resource group' (+ New), and 'Legal terms' (Review legal terms). The 'Parameters' tab contains a form with fields for ADMINUSERNAME (string), ADMINPASSWORD (securestring), DNSNAMEFORMASTERSPUBLICIP (string), JUMPBOXCONFIGURATION (string), DNSNAMEFORJUMPBOXPUBLICIP (string), NEWSTORAGEACCOUNTNAMEPREFIX (string), AGENTCOUNT (int), AGENTVM SIZE (string), MASTERCOUNT (int), MASTERVM SIZE (string), and MASTERCONFIGURATION (string). Numbered callouts 1 through 5 point to specific elements: 1 points to the Parameters form, 2 points to the OK button, 3 points to the Parameters section, 4 points to the Legal terms section, and 5 points to the Create button.

5. Fill out the form on the far right (marked “1” in the image), giving the following information. Important fields for which you will want to override the defaults are in bold:
  - (a) **AdminUsername**: Enter a username for logging into the cluster. It is easiest to set this to match your username on your local machine.
  - (b) **AdminPassword**: Choose a strong root password. Since you will be configuring SSH keys, you will not actually need to use this password to log in in practice, so choose something long and complex and store it safely.

- (c) **DnsNameForMastersPublicIp:** Enter a unique DNS name fragment to identify your cluster within your region. For example, if you are putting your cluster in `westus`, and you choose `awesomecluster`, your cluster's public IP would be assigned the name `awesomecluster.westus.cloudapp.azure.com`.
- (d) **JumpboxConfiguration:** If you would like, you can select to have either a Linux or Windows “jumpbox” with remote desktop software set up on the cluster's internal network. By default this is turned off, since it is unnecessary.
- (e) **DnsNameForJumpboxPublicIp:** If you are using a jumpbox, enter another unique DNS name fragment here to set its DNS name. See `DnsNameForMastersPublicIp` above.
- (f) **NewStorageAccountNamePrefix:** Enter a globally unique prefix to be used in the names of new storage accounts created to support the cluster. Storage account names must be 3 to 24 characters long, include only numbers and lower-case letters, and be globally unique. Since the template internally appends to this prefix, it must be shorter than the full 24 characters. Up to 20 should work.
- (g) **AgentCount:** Choose how many agents (i.e. worker nodes) you want in the cluster. Be mindful of your Azure subscription limits on both VMs (20 per region by default) and total cores (also 20 per region by default); if you ask for more agents or more total cores than you are allowed, you will not get them all, errors will occur during template instantiation, and the resulting cluster will be smaller than you wanted it to be.
- (h) **AgentVmSize:** Choose from the available VM instance sizes to determine how big each node will be. Again, be mindful of your Azure subscription's core limits. Also be mindful of how many cores and how much disk and memory your Toil jobs will need: if any requirement is greater than that provided by an entire node, a job may never be scheduled to run.
- (i) **MasterCount:** Choose the number of “masters” or leader nodes for the cluster. By default only one is used, because although the underlying Mesos batch system supports master failover, currently Toil does not. You can increase this if multiple Toil jobs will be running and you want them to run from different leader nodes. Remember that the leader nodes also count against your VM and core limits.
- (j) **MasterVmSize:** Select one of the available VM sizes to use for the leader nodes. Generally the leader node can be relatively small.
- (k) **MasterConfiguration:** This is set to `masters-are-not-agents` by default, meaning that the leader nodes will not themselves run any jobs. If you are worried about wasting unused computing power on your leader nodes, you can set this to `masters-are-agents` to allow them to run jobs. However, this may slow them down for interactive use, making it harder to monitor and control your Toil workflows.
- (l) **JumpboxVmSize:** If you are using a jumpbox, you can select a VM instance size for it to use here. Again, remember that it counts against your Azure subscription limits.
- (m) **ClusterPrefix:** This prefix gets used to generate the internal hostnames of all the machines in the cluster. You can use it to give clusters friendly names to differentiate them. It has to be a valid part of a DNS name; you might consider setting it to match `DnsNameForMastersPublicIp`. You can also leave it at the default.
- (n) **SwarmEnabled:** You can set this to `true` to install Swarm, a system for scheduling Docker containers. Toil does not use Swarm, and Swarm has a tendency to allocate all the cluster's resources for itself, so you should probably leave this set to `false` unless you also find yourself needing a Swarm cluster.
- (o) **MarathonEnabled:** You can set this to `true` to install Marathon, a scheduling system for persistent jobs run in Docker containers. It also has nothing to do with Toil, and should probably remain set to `false`.
- (p) **ChronosEnabled:** You can set this to `true` to install Chronos, which is a way to periodically run jobs on the cluster. Unless you find yourself needing this functionality, leave this set to `false`. (All these extra frameworks are here because the Toil Azure template was derived from a Microsoft template for a generic Mesos cluster, offering these services.)

- (q) **ToilEnabled**: You should leave this set to `true`. If you set it to `false`, Toil will not be installed on the cluster, which rather defeats the point.
  - (r) **SshRsaPublicKey**: Replace `default` with your SSH public key contents, beginning with `ssh-rsa`. Paste in the whole line. Only one key is supported, and as the name suggests it must be an RSA key. This enables SSH key-based login on the cluster.
  - (s) **GithubSource**: If you would like to install Toil from a nonstandard fork on Github (for example, installing a version including your own patches), set this to the Github fork (formatted as `<username>/<reponame>`) from which Toil should be downloaded and installed. If not, leave it set to the default of `BD2KGenomics/toil`.
  - (t) **GithubBranch**: To install Toil from a branch other than `master`, enter the name of its branch here. For example, for the latest release of Toil 3.1, enter `releases/3.1.x`. By default, you will get the latest and greatest Toil, but it may have bugs or breaking changes introduced since the last release.
6. Click OK (marked “2” in the screenshot).
  7. Choose a subscription and select or create a Resource Group (marked “3” in the screenshot). If creating a Resource Group, select a region in which to place it. It is recommended to create a new Resource Group for every cluster; the template creates a large number of Azure entities besides just the VMs (like virtual networks), and if they are organized into their own Resource Group they can all be cleaned up at once when you are done with the cluster, by deleting the Resource Group.
  8. Read the Azure terms of service (by clicking on the item marked “4” in the screenshot) and accept them by clicking the “Create” button on the right (not shown). This is the contract that you are accepting with Microsoft, under which you are purchasing the cluster.
  9. Click the main “Create” button (marked “5” in the screenshot). This will kick off the process of creating the cluster.
  10. Eventually you will receive a notification (Bell icon on the top bar of the Azure UI) letting you know that your cluster has been created. At this point, you should be able to connect to it; however, note that it will not be ready to run any Toil jobs until it is finished setting itself up.
  11. SSH into the first (and by default only) leader node. For this, you need to know the `AdminUsername` and `DnsNameForMastersPublicIp` you set above, and the name of the region you placed your cluster in. If you named your user `phoebe` and named your cluster `toilisgreat`, and placed it in the `centralus` region, the hostname of the cluster would be `toilisgreat.centralus.cloudapp.azure.com`, and you would want to connect as `phoebe`. SSH is forwarded through the cluster’s load balancer to the first leader node on port 2211, so you would run `ssh phoebe@toilisgreat.centralus.cloudapp.azure.com -p 2211`.
  12. Wait for the leader node to finish setting itself up. Run `tail -f /var/log/azure/cluster-bootstrap.log` and wait until the log reaches the line `completed mesos cluster configuration`. At that point, kill `tail` with a `ctrl-c`. Your leader node is now ready.
  13. At this point, you can start running Toil jobs, using the Mesos batch system (by passing `--batchSystem mesos --mesosMaster 10.0.0.5:5050`) and the Azure job store (for which you will need a separate Azure Storage account set up, ideally in the same region as your cluster but in a different Resource Group). The nodes of the cluster may take a few more minutes to finish installing, but when they do they will report in to Mesos and begin running any scheduled jobs.
  14. When you are done running your jobs, go back to the Azure portal, find the Resource Group you created for your cluster, and delete it. This will destroy all the VMs and any data stored on them, and stop Microsoft charging you money for keeping the cluster around. As long as you used a separate Azure Storage account in a different Resource Group, any information kept in the job stores and file stores you were using will be retained.

For more information about how your new cluster is organized, for information on how to access the Mesos Web UI, or for troubleshooting advice, please see [the template documentation](#).

## 2.3 Installation on OpenStack

Our group is working to expand distributed cluster support to OpenStack by providing convenient Docker containers to launch Mesos from. Currently, OpenStack nodes can be setup to run Toil in **singleMachine** mode following the basic installation instructions: *[Basic installation](#)*

## 2.4 Installation on Google Compute Engine

Support for running on Google Cloud is experimental, and our group is working to expand distributed cluster support to Google Compute by writing a cluster provisioning tool based around a Dockerized Mesos setup. Currently, Google Compute Engine nodes can be configured to run Toil in **singleMachine** mode following the basic installation instructions: *[Basic installation](#)*





---

## Running a workflow

---

### 3.1 Running quick start

Starting with Python, a Toil workflow can be run with just three steps.

1. `pip install toil`
2. Copy and paste the following code block into **HelloWorld.py**:

```
from toil.job import Job

def helloWorld(message, memory="2G", cores=2, disk="3G"):
    return "Hello, world!, here's a message: %s" % message

j = Job.wrapFn(helloWorld, "You did it!")

if __name__=="__main__":
    parser = Job.Runner.getDefaultArgumentParser()
    options = parser.parse_args()
    print Job.Runner.startToil(j, options) #Prints Hello, world!, ...
```

3. `python HelloWorld.py file:jobStore`

Now you have run Toil on **singleMachine** (default batch system) using the **FileStore** job store. The first positional argument after the `.py` is the location of the job store, a place where intermediate files are written to. In this example, a folder called **jobStore** will be created where **HelloWorld.py** is run from. Information on the jobStore can be found at [The job store interface](#).

Run `python HelloWorld.py --help` to see a complete list of available options.

For something beyond a hello world example, refer to [Running a Toil pipeline in detail](#)

### 3.2 Running CWL workflows

The **Common Workflow Language** (CWL) is an emerging standard for writing workflows that are portable across multiple workflow engines and platforms. To run workflows written using CWL, first ensure that Toil is installed with the “cwl” extra as described in [Basic installation](#). This will install the executables `cwl-runner` and `cwltoil` (these are identical, where `cwl-runner` is the portable name for the default system CWL runner). To learn more about CWL, see the [CWL User Guide](#).

To run in local batch mode, simply provide the CWL file and the input object file:

```
cwltoil example.cwl example-job.yml
```

To run in cloud and HPC configurations, you may need to provide additional command line parameters to select and configure the batch system to use. Consult the appropriate sections.

## 3.3 Running a Toil pipeline in detail

For a detailed example and explanation, we'll walk through running a pipeline that performs merge-sort on a temporary file.

1. Copy and paste the following code into **toil-sort-example.py**:

```
from __future__ import absolute_import
from argparse import ArgumentParser
import os
import logging
import random

from toil.job import Job

def setup(job, input_file, n, down_checkpoints):
    """Sets up the sort.
    """
    # Write the input file to the file store
    input_filestore_id = job.fileStore.writeGlobalFile(input_file, True)
    job.fileStore.logToMaster(" Starting the merge sort ")
    job.addFollowOnJobFn(cleanup, job.addChildJobFn(down,
                                                    input_filestore_id, n,
                                                    down_checkpoints=down_checkpoints,
                                                    memory='1000M').rv(), input_file)

def down(job, input_file_store_id, n, down_checkpoints):
    """Input is a file and a range into that file to sort and an output location in which
    to write the sorted file.
    If the range is larger than a threshold N the range is divided recursively and
    a follow on job is then created which merges back the results else
    the file is sorted and placed in the output.
    """
    # Read the file
    input_file = job.fileStore.readGlobalFile(input_file_store_id, cache=False)
    length = os.path.getsize(input_file)
    if length > n:
        # We will subdivide the file
        job.fileStore.logToMaster("Splitting file: %s of size: %s"
                                  % (input_file_store_id, length), level=logging.CRITICAL)
        # Split the file into two copies
        mid_point = get_midpoint(input_file, 0, length)
        t1 = job.fileStore.getLocalTempFile()
        with open(t1, 'w') as fh:
            copy_subrange_of_file(input_file, 0, mid_point + 1, fh)
        t2 = job.fileStore.getLocalTempFile()
        with open(t2, 'w') as fh:
            copy_subrange_of_file(input_file, mid_point + 1, length, fh)
        # Call down recursively
```

```

        return job.addFollowOnJobFn(up, job.addChildJobFn(down, job.fileStore.writeGlobalFile(t1),
        down_checkpoints=down_checkpoints, memory='1000M').rv(),
        job.addChildJobFn(down, job.fileStore.writeGlobalFile(t2), n
        down_checkpoints=down_checkpoints,
        memory='1000M').rv()).rv())

    else:
        # We can sort this bit of the file
        job.fileStore.logToMaster("Sorting file: %s of size: %s"
        % (input_file_store_id, length), level=logging.CRITICAL)
        # Sort the copy and write back to the fileStore
        sort(input_file)
        return job.fileStore.writeGlobalFile(input_file)

def up(job, input_file_id_1, input_file_id_2):
    """Merges the two files and places them in the output.
    """
    with job.fileStore.writeGlobalFileStream() as (fileHandle, output_id):
        with job.fileStore.readGlobalFileStream(input_file_id_1) as inputFileHandle1:
            with job.fileStore.readGlobalFileStream(input_file_id_2) as inputFileHandle2:
                merge(inputFileHandle1, inputFileHandle2, fileHandle)
                job.fileStore.logToMaster("Merging %s and %s to %s"
                % (input_file_id_1, input_file_id_2, output_id))
        # Cleanup up the input files - these deletes will occur after the completion is successful
        job.fileStore.deleteGlobalFile(input_file_id_1)
        job.fileStore.deleteGlobalFile(input_file_id_2)
        return output_id

def cleanup(job, temp_output_id, output_file):
    """Copies back the temporary file to input once we've successfully sorted the temporary file
    """
    job.fileStore.readGlobalFile(temp_output_id, userPath=output_file)
    job.fileStore.logToMaster("Finished copying sorted file to output: %s" % output_file)

# convenience functions
def sort(file):
    """Sorts the given file.
    """
    filehandle = open(file, 'r')
    lines = filehandle.readlines()
    filehandle.close()
    lines.sort()
    filehandle = open(file, 'w')
    for line in lines:
        filehandle.write(line)
    filehandle.close()

def merge(filehandle_1, filehandle_2, output_filehandle):
    """Merges together two files maintaining sorted order.
    """
    line2 = filehandle_2.readline()
    for line1 in filehandle_1.readlines():
        while line2 != '' and line2 <= line1:
            output_filehandle.write(line2)
            line2 = filehandle_2.readline()

```

```
        output_filehandle.write(line1)
    while line2 != '':
        output_filehandle.write(line2)
        line2 = filehandle_2.readline()

def copy_subrange_of_file(input_file, file_start, file_end, output_filehandle):
    """Copies the range (in bytes) between fileStart and fileEnd to the given
    output file handle.
    """
    with open(input_file, 'r') as fileHandle:
        fileHandle.seek(file_start)
        data = fileHandle.read(file_end - file_start)
        assert len(data) == file_end - file_start
        output_filehandle.write(data)

def get_midpoint(file, file_start, file_end):
    """Finds the point in the file to split.
    Returns an int i such that fileStart <= i < fileEnd
    """
    filehandle = open(file, 'r')
    mid_point = (file_start + file_end) / 2
    assert mid_point >= file_start
    filehandle.seek(mid_point)
    line = filehandle.readline()
    assert len(line) >= 1
    if len(line) + mid_point < file_end:
        return mid_point + len(line) - 1
    filehandle.seek(file_start)
    line = filehandle.readline()
    assert len(line) >= 1
    assert len(line) + file_start <= file_end
    return len(line) + file_start - 1

def make_file_to_sort(file_name, lines, line_length):
    with open(file_name, 'w') as fileHandle:
        for _ in xrange(lines):
            line = "".join(random.choice('actgACTGNXYZ') for _ in xrange(line_length - 1)) + '\n'
            fileHandle.write(line)

def main():
    parser = ArgumentParser()
    Job.Runner.addToilOptions(parser)

    parser.add_argument('--num-lines', default=1000, help='Number of lines in file to sort.', type=int)
    parser.add_argument('--line-length', default=50, help='Length of lines in file to sort.', type=int)
    parser.add_argument("--N",
                        help="The threshold below which a serial sort function is used to sort file. If N is 0, then a merge sort function is used. All lines must of length less than or equal to N or program will fail",
                        default=10000)

    options = parser.parse_args()

    if int(options.N) <= 0:
        raise RuntimeError("Invalid value of N: %s" % options.N)
```

```

        make_file_to_sort(file_name='file_to_sort.txt', lines=options.num_lines, line_length=options.

    # Now we are ready to run
    Job.Runner.startToil(Job.wrapJobFn(setup, 'file_to_sort.txt', int(options.N), False,
                                     memory='1000M'), options)

if __name__ == '__main__':
    main()

```

2. Run with default settings: `python toil-sort-example.py file:jobStore`.

3. Run with options: `python toil-sort-example.py file:jobStore --num-lines 5000 --line-length 10 --workDir /tmp/`

The `if __name__ == '__main__':` boilerplate is required to enable Toil to import the job functions defined in the script into the context of a Toil *worker* process. By invoking the script you created the *leader process*. A worker process is a separate process whose sole purpose is to host the execution of one or more jobs defined in that script. When using the single-machine batch system (the default), the worker processes will be running on the same machine as the leader process. With full-fledged batch systems like Mesos the worker processes will typically be started on separate machines. The boilerplate ensures that the pipeline is only started once—on the leader—but not when its job functions are imported and executed on the individual workers.

Typing `python toil-sort-example.py --help` will show the complete list of arguments for the workflow which includes both Toil's and ones defined inside **toil-sort-example.py**. A complete explanation of Toil's arguments can be found in [Command line interface and arguments](#).

### 3.3.1 Changing the log statements

When we run the pipeline, we see some logs printed to the screen. At the top there's some information provided to the user about the environment Toil is being setup in, and then as the pipeline runs we get INFO level messages from the batch system that tell us when jobs are being executed. We also see both INFO and CRITICAL level messages that are in the user script. By changing the `logLevel`, we can change what we see output to screen. For only CRITICAL level messages: `python toil-sort-example.py file:jobStore --logLevel=critical`. This hides most of the information we get from the Toil run. For more detail, we can run the pipeline with `--logLevel=debug` to see a comprehensive output. For more information see [Logging](#).

### 3.3.2 Restarting after introducing a bug

Let's now introduce a bug in the code, so we can understand what a failure looks like in Toil, and how we would go about resuming the pipeline. On line 30, the first line of the `down()` function, let's add the line `assert 1==2, 'Test Error!'`. Now when we run the pipeline, `python toil-sort-example.py file:jobStore`, we'll see a failure log under the header `--- TOIL WORKER OUTPUT LOG---`, that contains the stack trace. We see a detailed message telling us that on line 30, in the `down` function, we encountered an error.

If we try and run the pipeline again, we get an error message telling us that a `jobStore` of the same name already exists. The default behavior for the job store is that it is not cleaned up in the event of failure so that you can restart it from the last successful job. We can restart the pipeline by running `python toil-sort-example.py file:jobStore --restart`. We can also change the number of times Toil will attempt to retry a failed job, `python toil-sort-example.py --retryCount 2 --restart`. You'll now see Toil attempt to rerun the failed job, decrementing a counter until that job has exhausted the retry count. `--retryCount` is useful for non-systemic errors, like downloading a file that may experience a sporadic interruption, or some other non-deterministic failure.

To successfully restart our pipeline, we can edit our script to comment out line 30, or remove it, and then run `python toil-sort-example.py --restart`. The pipeline will successfully complete, and the job store will be removed.

### 3.3.3 Getting stats from our pipeline run

We can execute the pipeline to let us retrieve statistics with `python toil-sort-example.py --stats`. Our pipeline will finish successfully, but leave behind the job store. Now we can type `toil stats file:jobStore` and get back information about total runtime and stats pertaining to each job function.

We can then cleanup our jobStore by running `toil clean file:jobStore`

---

## Running in the cloud

---

There are several recommended ways to run Toil jobs in the cloud. Of these, running on Amazon Web Services (AWS) is currently the best-supported solution.

On all cloud providers, it is recommended that you run long-running jobs on remote systems under `screen`. Simply type `screen` to open a new `screen` session. Later, type `Ctrl-a` and then `d` to disconnect from it, and run `screen -r` to reconnect to it. Commands running under `screen` will continue running even when you are disconnected, allowing you to unplug your laptop and take it home without ending your Toil jobs.

### 4.1 Running on AWS

See *Installation on AWS for distributed computing* to get setup for running on AWS.

Having followed the *Running quick start* guide, the user can run their **HelloWorld.py** script on a distributed cluster just by modifying the run command. Since our cluster is distributed, we'll use the **AWS Jobstore** which creates a job store in S3 instead of on file system.

Place the HelloWorld.py script on the leader node, and run:

```
python --batchSystem=mesos --mesosMaster=mesos-master:5050 \  
      HelloWorld.py aws:us-west-2:my-s3-jobstore
```

To run a CWL workflow:

```
cwltoil --batchSystem=mesos --mesosMaster=mesos-master:5050 \  
      --jobStore=aws:us-west-2:my-s3-jobstore \  
      example.cwl example-job.yml
```

When running a CWL workflow on AWS, input files can be provided either on the local file system or in S3 buckets using `s3://` URL references. Final output files will be copied to the local file system of the leader node.

### 4.2 Running on Azure

See *Installation on Azure* to get setup for running on Azure. This section assumes that you are SSHed into your cluster's leader node.

The Azure templates do not create a shared filesystem; you need to use the **Azure Jobstore**, which needs an Azure Storage Account in which to store its job data. (Note that you can store multiple job stores in a single Azure Storage Account.)

To create a new Storage Account, if you do not already have one:

1. [Click here](https://portal.azure.com/#create/Microsoft.StorageAccount), or navigate to `https://portal.azure.com/#create/Microsoft.StorageAccount` in your browser.
2. If necessary, log into the Microsoft Account that you use for Azure.
3. Fill out the presented form. The **Name** for the account, notably, must be a 3-to-24-character string of letters and lowercase numbers that is globally unique. For **Deployment model**, choose “Resource manager”. For **Resource group**, choose or create a resource group **different than** the one in which you created your cluster. For **Location**, choose the **same** region that you used for your cluster.
4. Press the “Create” button. Wait for your Storage Account to be created; you should get a notification in the notifications area at the upper right.

Once you have a Storage Account, you need to authorize the cluster to access the Storage Account, by giving it the access key. To do find your Storage Account’s access key:

1. When your Storage Account has been created, open it up and click the “Settings” icon.
2. In the “Settings” panel, select “Access keys”.
3. Select the text in the “Key1” box and copy it to the clipboard, or use the copy-to-clipboard icon.

You then need to share the key with the cluster. To do this temporarily, for the duration of an SSH or screen session:

1. On the leader node, run `export AZURE_ACCOUNT_KEY="<KEY>"`, replacing `<KEY>` with the access key you copied from the Azure portal.

To do this permanently:

1. On the leader node, run `nano ~/.toilAzureCredentials`.
2. In the editor that opens, navigate with the arrow keys, and give the file the following contents:

```
[AzureStorageCredentials]
<accountname>=<accountkey>
```

Be sure to replace `<accountname>` with the name that you used for your Azure Storage Account, and `<accountkey>` with the key you obtained above. (If you want, you can have multiple accounts with different keys in this file, by adding multiple lines. If you do this, be sure to leave the `AZURE_ACCOUNT_KEY` environment variable unset.)

3. Press `ctrl-o` to save the file, and `ctrl-x` to exit the editor.

Once that’s done, you are now ready to actually execute a job, storing your job store in that Azure Storage Account. Assuming you followed the *Running quick start* guide above, you have an Azure Storage Account created, and you have placed the Storage Account’s access key on the cluster, you can run the **HelloWorld.py** script by doing the following:

1. Place your script on the leader node, either by downloading it from the command line or typing or copying it into a command-line editor.
2. Run the command:

```
python --batchSystem=mesos --mesosMaster=10.0.0.5:5050 \
    HelloWorld.py azure:<accountname>:hello-world001
```

To run a CWL workflow:

```
cwltoil --batchSystem=mesos --mesosMaster=10.0.0.5:5050 \
    --jobStore=azure:<accountname>:hello-world001 \
    example.cwl example-job.yml
```

Be sure to replace `<accountname>` with the name of your Azure Storage Account.



Note that once you run a job with a particular job store name (the part after the account name) in a particular Storage Account, you cannot re-use that name in that account unless one of the following happens:

1. You are restarting the same job with the `--restart` option.
2. You clean the job store with `toil clean azure:<accountname>:<jobstore>`.
3. You delete all the items created by that job, and the main job store table used by Toil, from the account (destroying all other job stores using the account).
4. The job finishes successfully and cleans itself up.

## 4.3 Running on Open Stack

After getting setup with *Installation on OpenStack*, Toil scripts can be run just by designating a job store location as shown in *Running quick start*. The location of temporary directories Toil creates to run jobs can be specified with `--workDir`:

```
python HelloWorld.py file:jobStore --workDir /tmp/
```

## 4.4 Running on Google Compute Engine

After getting setup with *Installation on Google Compute Engine*, Toil scripts can be run just by designating a job store location as shown in *Running quick start*.

If you wish to use the Google Storage job store, you must install Toil with the ‘google’ extra. Having done this, you must create a file named ‘.boto’ in your home directory with the following format:

```
[Credentials]
gs_access_key_id = KEY_ID
gs_secret_access_key = SECRET_KEY

[Boto]
https_validate_certificates = True

[GSUtil]
content_language = en
default_api_version = 2
```

The `gs_access_key_id` and `gs_secret_access_key` can be generated by navigating to your Google Cloud Storage console and clicking on ‘Settings’. Then, on the Settings page, navigate to the Interoperability tab and click ‘Enable interoperability access’. On this page you can now click ‘Create a new key’ to generate an access key and a matching secret. Insert these into their respective places in the .boto file and you will be able to use a Google job store when invoking a Toil script, as in the following example:

```
python HelloWorld.py google:projectID:jobStore
```

The ‘projectID’ component of the job store argument above refers your Google Cloud project ID in the Google Cloud Console, and will be visible in the console’s banner at the top of the screen. The ‘jobStore’ component is a name of your choosing that you will use to refer to this job store.



---

## Command line interface and arguments

---

Toil provides many command line options when running a toil script (see [Running a workflow](#)), or using Toil to run a CWL or WDL script. Many of these are described below. For most Toil scripts executing ‘`--help`’ will show this list of options.

It is also possible to set and manipulate the options described when invoking a Toil workflow from within Python using `toil.job.Job.Runner.getDefaultOptions()`, e.g.:

```
options = Job.Runner.getDefaultOptions("./toilWorkflow") # Get the options object
options.logLevel = "INFO" # Set the log level to the info level.

Job.Runner.startToil(Job(), options) # Run the script
```

### 5.1 Logging

Toil hides stdout and stderr by default except in case of job failure. For more robust logging options (default is INFO), use `--logDebug` or more generally, use `--logLevel=`, which may be set to either OFF (or CRITICAL), ERROR, WARN (or WARNING), INFO or DEBUG. Logs can be directed to a file with `--logFile=`.

If large logfiles are a problem, `--maxLogFileSize` (in bytes) can be set as well as `--rotatingLogging`, which prevents logfiles from getting too large.

### 5.2 Stats

The `--stats` argument records statistics about the Toil workflow in the job store. After a Toil run has finished, the entrypoint `toil stats <jobStore>` can be used to return statistics about cpu, memory, job duration, and more. The job store will never be deleted with `--stats`, as it overrides `--clean`.

### 5.3 Restart

In the event of failure, Toil can resume the pipeline by adding the argument `--restart` and rerunning the python script. Toil pipelines can even be edited and resumed which is useful for development or troubleshooting.

## 5.4 Clean

If a Toil pipeline didn't finish successfully, or is using a variation of `--clean`, the job store will exist until it is deleted. `toil clean <jobStore>` ensures that all artifacts associated with a job store are removed. This is particularly useful for deleting AWS job stores, which reserves an SDB domain as well as an S3 bucket.

The deletion of the job store can be modified by the `--clean` argument, and may be set to `always`, `onError`, `never`, or `onSuccess` (default).

Temporary directories where jobs are running can also be saved from deletion using the `--cleanWorkDir`, which has the same options as `--clean`. This option should only be run when debugging, as intermediate jobs will fill up disk space.

## 5.5 Batch system

Toil supports several different batch systems using the `--batchSystem` argument. More information in the *The batch system interface*.

## 5.6 Default cores, disk, and memory

Toil uses resource requirements to intelligently schedule jobs. The defaults for cores (1), disk (2G), and memory (2G), can all be changed using `--defaultCores`, `--defaultDisk`, and `--defaultMemory`. Standard suffixes like K, Ki, M, Mi, G or Gi are supported.

## 5.7 Job store

Running toil scripts has one required positional argument: the job store. The default job store is just a path to where the user would like the job store to be created. To use the *Running quick start* example, if you're on a node that has a large **/scratch** volume, you can specify the jobstore be created there by executing: `python HelloWorld.py /scratch/my-job-store`, or more explicitly, `python HelloWorld.py file:/scratch/my-job-store`. Toil uses the colon as way to explicitly name what type of job store the user would like. Different types of job store options can be looked up in *The job store interface*.

## 5.8 Miscellaneous

Here are some additional useful arguments that don't fit into another category.

- `--workDir` sets the location where temporary directories are created for running jobs.
- `--retryCount` sets the number of times to retry a job in case of failure. Useful for non-systemic failures like HTTP requests.
- `--sseKey` accepts a path to a 32-byte key that is used for server-side encryption when using the AWS job store.
- `--cseKey` accepts a path to a 256-bit key to be used for client-side encryption on Azure job store.
- `--setEnv <NAME=VALUE>` sets an environment variable early on in the worker

---

## Developing a workflow

---

This tutorial walks through the features of Toil necessary for developing a workflow using the Toil Python API.

### 6.1 Scripting quick start

To begin, consider this short toil script which illustrates defining a workflow:

```
from toil.job import Job

def helloWorld(message, memory="2G", cores=2, disk="3G"):
    return "Hello, world!, here's a message: %s" % message

j = Job.wrapFn(helloWorld, "woot")

if __name__=="__main__":
    options = Job.Runner.getDefaultOptions("./toilWorkflow")
    print Job.Runner.startToil(j, options) #Prints Hello, world!, ...
```

The workflow consists of a single job. The resource requirements for that job are (optionally) specified by keyword arguments (memory, cores, disk). The script is run using `toil.job.Job.Runner.getDefaultOptions()`. Below we explain the components of this code in detail.

### 6.2 Job basics

The atomic unit of work in a Toil workflow is a *job* (`toil.job.Job`). User scripts inherit from this base class to define units of work. For example, here is a more long-winded class-based version of the job in the quick start example:

```
from toil.job import Job

class HelloWorld(Job):
    def __init__(self, message):
        Job.__init__(self, memory="2G", cores=2, disk="3G")
        self.message = message

    def run(self, fileStore):
        return "Hello, world!, here's a message: %s" % self.message
```

In the example a class, `HelloWorld`, is defined. The constructor requests 2 gigabytes of memory, 2 cores and 3 gigabytes of local disk to complete the work.

The `toil.job.Job.run()` method is the function the user overrides to get work done. Here it just logs a message using `toil.job.Job.FileStore.logToMaster()`, which will be registered in the log output of the leader process of the workflow.

## 6.3 Invoking a workflow

We can add to the previous example to turn it into a complete workflow by adding the necessary function calls to create an instance of `HelloWorld` and to run this as a workflow containing a single job. This uses the `toil.job.Job.Runner` class, which is used to start and resume Toil workflows. For example:

```
from toil.job import Job

class HelloWorld(Job):
    def __init__(self, message):
        Job.__init__(self, memory="2G", cores=2, disk="3G")
        self.message = message

    def run(self, fileStore):
        return "Hello, world!, here's a message: %s" % self.message

if __name__=="__main__":
    options = Job.Runner.getDefaultOptions("./toilWorkflowRun")
    print Job.Runner.startToil(HelloWorld("woot"), options)
```

Alternatively, the more powerful `toil.common.Toil` class can be used to run and resume workflows. It is used as a context manager and allows for preliminary setup, such as staging of files into the job store on the leader node. An instance of the class is initialized by specifying an options object. The actual workflow is then invoked by calling the `toil.common.Toil.start()` method, passing the root job of the workflow, or, if a workflow is being restarted, `toil.common.Toil.restart()` should be used. Note that the context manager should have explicit if else branches addressing restart and non restart cases. The boolean value for these if else blocks is `toil.options.restart`.

For example:

```
from toil.job import Job
from toil.common import Toil

class HelloWorld(Job):
    def __init__(self, message):
        Job.__init__(self, memory="2G", cores=2, disk="3G")
        self.message = message

    def run(self, fileStore):
        fileStore.logToMaster("Hello, world!, I have a message: %s"
                              % self.message)

if __name__=="__main__":
    options = Job.Runner.getDefaultOptions("./toilWorkflowRun")
    options.logLevel = "INFO"

    with Toil(options) as toil:
        if not toil.options.restart:
            job = HelloWorld("Smitty Werbenmanjensen, he was #1")
            toil.start(job)
        else:
            toil.restart()
```

The call to `toil.job.Job.Runner.getDefaultOptions()` creates a set of default options for the workflow. The only argument is a description of how to store the workflow's state in what we call a *job-store*. Here the job-store

is contained in a directory within the current working directory called “toilWorkflowRun”. Alternatively this string can encode other ways to store the necessary state, e.g. an S3 bucket or Azure object store location. By default the job-store is deleted if the workflow completes successfully.

The workflow is executed in the final line, which creates an instance of `HelloWorld` and runs it as a workflow. Note all Toil workflows start from a single starting job, referred to as the *root* job. The return value of the root job is returned as the result of the completed workflow (see promises below to see how this is a useful feature!).

## 6.4 Specifying arguments via the command line

To allow command line control of the options we can use the `toil.job.Job.Runner.getDefaultArgumentParser()` method to create a `argparse.ArgumentParser` object which can be used to parse command line options for a Toil script. For example:

```
from toil.job import Job

class HelloWorld(Job):
    def __init__(self, message):
        Job.__init__(self, memory="2G", cores=2, disk="3G")
        self.message = message

    def run(self, fileStore):
        return "Hello, world!, here's a message: %s" % self.message

if __name__ == "__main__":
    parser = Job.Runner.getDefaultArgumentParser()
    options = parser.parse_args()
    print Job.Runner.startToil(HelloWorld("woot"), options)
```

Creates a fully fledged script with all the options Toil exposed as command line arguments. Running this script with “--help” will print the full list of options.

Alternatively an existing `argparse.ArgumentParser` or `optparse.OptionParser` object can have Toil script command line options added to it with the `toil.job.Job.Runner.addToilOptions()` method.

## 6.5 Resuming a workflow

In the event that a workflow fails, either because of programmatic error within the jobs being run, or because of node failure, the workflow can be resumed. Workflows can only not be reliably resumed if the job-store itself becomes corrupt.

Critical to resumption is that jobs can be rerun, even if they have apparently completed successfully. Put succinctly, a user defined job should not corrupt its input arguments. That way, regardless of node, network or leader failure the job can be restarted and the workflow resumed.

To resume a workflow specify the “restart” option in the options object passed to `toil.job.Job.Runner.startToil()`. If node failures are expected it can also be useful to use the integer “retryCount” option, which will attempt to rerun a job retryCount number of times before marking it fully failed.

In the common scenario that a small subset of jobs fail (including retry attempts) within a workflow Toil will continue to run other jobs until it can do no more, at which point `toil.job.Job.Runner.startToil()` will raise a `toil.job.leader.FailedJobsException` exception. Typically at this point the user can decide to fix the script and resume the workflow or delete the job-store manually and rerun the complete workflow.

## 6.6 Functions and job functions

Defining jobs by creating class definitions generally involves the boilerplate of creating a constructor. To avoid this the classes `toil.job.FunctionWrappingJob` and `toil.job.JobFunctionWrappingTarget` allow functions to be directly converted to jobs. For example, the quick start example (repeated here):

```
from toil.job import Job

def helloWorld(message, memory="2G", cores=2, disk="3G"):
    return "Hello, world!, here's a message: %s" % message

j = Job.wrapFn(helloWorld, "woot")

if __name__=="__main__":
    options = Job.Runner.getDefaultOptions("./toilWorkflowRun")
    print Job.Runner.startToil(j, options)
```

Is equivalent to the previous example, but using a function to define the job.

The function call:

```
Job.wrapFn(helloWorld, "woot")
```

Creates the instance of the `toil.job.FunctionWrappingTarget` that wraps the function.

The keyword arguments *memory*, *cores* and *disk* allow resource requirements to be specified as before. Even if they are not included as keyword arguments within a function header they can be passed as arguments when wrapping a function as a job and will be used to specify resource requirements.

We can also use the function wrapping syntax to a *job function*, a function whose first argument is a reference to the wrapping job. Just like a *self* argument in a class, this allows access to the methods of the wrapping job, see `toil.job.JobFunctionWrappingTarget`. For example:

```
from toil.job import Job

def helloWorld(job, message):
    job.fileStore.logToMaster("Hello world, "
    "I have a message: %s" % message) # This uses a logging function
    # of the Job.FileStore class

if __name__=="__main__":
    options = Job.Runner.getDefaultOptions("./toilWorkflowRun")
    options.logLevel = "INFO"
    print Job.Runner.startToil(Job.wrapJobFn(helloWorld, "woot"), options)
```

Here `helloWorld2` is a job function. It accesses the `toil.job.Job.FileStore` attribute of the job to log a message that will be printed to the output console. Here the only subtle difference to note is the line:

```
Job.Runner.startToil(Job.wrapJobFn(helloWorld, "woot"), options)
```

Which uses the function `toil.job.Job.wrapJobFn()` to wrap the job function instead of `toil.job.Job.wrapFn()` which wraps a vanilla function.

## 6.7 Workflows with multiple jobs

A *parent* job can have *child* jobs and *follow-on* jobs. These relationships are specified by methods of the job class, e.g. `toil.job.Job.addChild()` and `toil.job.Job.addFollowOn()`.



Considering a set of jobs the nodes in a job graph and the child and follow-on relationships the directed edges of the graph, we say that a job B that is on a directed path of child/follow-on edges from a job A in the job graph is a *successor* of A, similarly A is a *predecessor* of B.

A parent job's child jobs are run directly after the parent job has completed, and in parallel. The follow-on jobs of a job are run after its child jobs and their successors have completed. They are also run in parallel. Follow-ons allow the easy specification of cleanup tasks that happen after a set of parallel child tasks. The following shows a simple example that uses the earlier `helloWorld` job function:

```
from toil.job import Job

def helloWorld(job, message, memory="2G", cores=2, disk="3G"):
    job.fileStore.logToMaster("Hello world, "
    "I have a message: %s" % message) # This uses a logging function
    # of the Job.FileStore class

j1 = Job.wrapJobFn(helloWorld, "first")
j2 = Job.wrapJobFn(helloWorld, "second or third")
j3 = Job.wrapJobFn(helloWorld, "second or third")
j4 = Job.wrapJobFn(helloWorld, "last")
j1.addChild(j2)
j1.addChild(j3)
j1.addFollowOn(j4)

if __name__=="__main__":
    options = Job.Runner.getDefaultOptions("./toilWorkflowRun")
    options.logLevel = "INFO"
    Job.Runner.startToil(j1, options)
```

In the example four jobs are created, first `j1` is run, then `j2` and `j3` are run in parallel as children of `j1`, finally `j4` is run as a follow-on of `j1`.

There are multiple short hand functions to achieve the same workflow, for example:

```
from toil.job import Job

def helloWorld(job, message, memory="2G", cores=2, disk="3G"):
    job.fileStore.logToMaster("Hello world, "
    "I have a message: %s" % message) # This uses a logging function
    # of the Job.FileStore class

j1 = Job.wrapJobFn(helloWorld, "first")
j2 = j1.addChildJobFn(helloWorld, "second or third")
j3 = j1.addChildJobFn(helloWorld, "second or third")
j4 = j1.addFollowOnJobFn(helloWorld, "last")

if __name__=="__main__":
    options = Job.Runner.getDefaultOptions("./toilWorkflowRun")
    options.logLevel = "INFO"
    Job.Runner.startToil(j1, options)
```

Equivalently defines the workflow, where the functions `toil.job.Job.addChildJobFn()` and `toil.job.Job.addFollowOnJobFn()` are used to create job functions as children or follow-ons of an earlier job.

Jobs graphs are not limited to trees, and can express arbitrary directed acyclic graphs. For a precise definition of legal graphs see `toil.job.Job.checkJobGraphForDeadlocks()`. The previous example could be specified as a DAG as follows:

```
from toil.job import Job

def helloWorld(job, message, memory="2G", cores=2, disk="3G"):
    job.fileStore.logToMaster("Hello world, "
    "I have a message: %s" % message) # This uses a logging function
    # of the Job.FileStore class

j1 = Job.wrapJobFn(helloWorld, "first")
j2 = j1.addChildJobFn(helloWorld, "second or third")
j3 = j1.addChildJobFn(helloWorld, "second or third")
j4 = j2.addChildJobFn(helloWorld, "last")
j3.addChild(j4)

if __name__=="__main__":
    options = Job.Runner.getDefaultOptions("./toilWorkflowRun")
    options.logLevel = "INFO"
    Job.Runner.startToil(j1, options)
```

Note the use of an extra child edge to make j4 a child of both j2 and j3.

## 6.8 Dynamic job creation

The previous examples show a workflow being defined outside of a job. However, Toil also allows jobs to be created dynamically within jobs. For example:

```
from toil.job import Job

def binaryStringFn(job, message="", depth):
    if depth > 0:
        job.addChildJobFn(binaryStringFn, message + "0", depth-1)
        job.addChildJobFn(binaryStringFn, message + "1", depth-1)
    else:
        job.fileStore.logToMaster("Binary string: %s" % message)

if __name__=="__main__":
    options = Job.Runner.getDefaultOptions("./toilWorkflowRun")
    options.logLevel = "INFO"
    Job.Runner.startToil(Job.wrapJobFn(binaryStringFn, depth=5), options)
```

The `binaryStringFn` logs all possible binary strings of length  $n$  (here  $n=5$ ), creating a total of  $2^{(n+2)} - 1$  jobs dynamically and recursively. Static and dynamic creation of jobs can be mixed in a Toil workflow, with jobs defined within a job or job function being created at run-time.

## 6.9 Promises

The previous example of dynamic job creation shows variables from a parent job being passed to a child job. Such forward variable passing is naturally specified by recursive invocation of successor jobs within parent jobs. This can also be achieved statically by passing around references to the return variables of jobs. In Toil this is achieved with promises, as illustrated in the following example:

```
from toil.job import Job

def fn(job, i):
    job.fileStore.logToMaster("i is: %s" % i, level=100)
```

```

    return i+1

j1 = Job.wrapJobFn(fn, 1)
j2 = j1.addChildJobFn(fn, j1.rv())
j3 = j1.addFollowOnJobFn(fn, j2.rv())

if __name__=="__main__":
    options = Job.Runner.getDefaultOptions("./toilWorkflowRun")
    options.logLevel = "INFO"
    Job.Runner.startToil(j1, options)

```

Running this workflow results in three log messages from the jobs: “i is 1” from *j1*, “i is 2” from *j2* and “i is 3” from *j3*.

The return value from the first job is *promised* to the second job by the call to `toil.job.Job.rv()` in the line:

```
j2 = j1.addChildFn(fn, j1.rv())
```

The value of `j1.rv()` is a *promise*, rather than the actual return value of the function, because *j1* for the given input has at that point not been evaluated. A promise (`toil.job.Promise`) is essentially a pointer to the return value that is replaced by the actual return value once it has been evaluated. Therefore when *j2* is run the promise becomes 2.

Promises can be quite useful. For example, we can combine dynamic job creation with promises to achieve a job creation process that mimics the functional patterns possible in many programming languages:

```

from toil.job import Job

def binaryStrings(job, message="", depth):
    if depth > 0:
        s = [ job.addChildJobFn(binaryStrings, message + "0",
                                depth-1).rv(),
              job.addChildJobFn(binaryStrings, message + "1",
                                depth-1).rv() ]
        return job.addFollowOnFn(merge, s).rv()
    return [message]

def merge(strings):
    return strings[0] + strings[1]

if __name__=="__main__":
    options = Job.Runner.getDefaultOptions("./toilWorkflowRun")
    l = Job.Runner.startToil(Job.wrapJobFn(binaryStrings, depth=5), options)
    print l #Prints a list of all binary strings of length 5

```

The return value *l* of the workflow is a list of all binary strings of length 10, computed recursively. Although a toy example, it demonstrates how closely Toil workflows can mimic typical programming patterns.

## 6.10 Managing files within a workflow

It is frequently the case that a workflow will want to create files, both persistent and temporary, during its run. The `toil.job.Job.FileStore` class is used by jobs to manage these files in a manner that guarantees cleanup and resumption on failure.

The `toil.job.Job.run()` method has a file-store instance as an argument. The following example shows how this can be used to create temporary files that persist for the length of the job, be placed in a specified local disk of the node and that will be cleaned up, regardless of failure, when the job finishes:

```
from toil.job import Job

class LocalFileStoreJob(Job):
    def run(self, fileStore):
        scratchDir = fileStore.getLocalTempDir() #Create a temporary
        # directory safely within the allocated disk space
        # reserved for the job.

        scratchFile = fileStore.getLocalTempFile() #Similarly
        # create a temporary file.

if __name__=="__main__":
    options = Job.Runner.getDefaultOptions("./toilWorkflowRun")
    #Create an instance of FooJob which will
    # have at least 10 gigabytes of storage space.
    j = LocalFileStoreJob(disk="10G")
    #Run the workflow
    Job.Runner.startToil(j, options)
```

Job functions can also access the file-store for the job. The equivalent of the `LocalFileStoreJob` class is equivalently:

```
def localFileStoreJobFn(job):
    scratchDir = job.fileStore.getLocalTempDir()
    scratchFile = job.fileStore.getLocalTempFile()
```

Note that the `fileStore` attribute is accessed as an attribute of the job argument.

In addition to temporary files that exist for the duration of a job, the file-store allows the creation of files in a *global* store, which persists during the workflow and are globally accessible (hence the name) between jobs. For example:

```
from toil.job import Job
import os

def globalFileStoreJobFn(job):
    job.fileStore.logToMaster("The following example exercises all the"
                              " methods provided by the"
                              " Job.FileStore class")

    scratchFile = job.fileStore.getLocalTempFile() # Create a local
    # temporary file.

    with open(scratchFile, 'w') as fH: # Write something in the
    # scratch file.
        fH.write("What a tangled web we weave")

    # Write a copy of the file into the file-store;
    # fileID is the key that can be used to retrieve the file.
    fileID = job.fileStore.writeGlobalFile(scratchFile) #This write
    # is asynchronous by default

    # Write another file using a stream; fileID2 is the
    # key for this second file.
    with job.fileStore.writeGlobalFileStream(cleanup=True) as (fH, fileID2):
        fH.write("Out brief candle")

    # Now read the first file; scratchFile2 is a local copy of the file
    # that is read only by default.
    scratchFile2 = job.fileStore.readGlobalFile(fileID)
```

```

# Read the second file to a desired location: scratchFile3.
scratchFile3 = os.path.join(job.fileStore.getLocalTempDir(), "foo.txt")
job.fileStore.readGlobalFile(fileID, userPath=scratchFile3)

# Read the second file again using a stream.
with job.fileStore.readGlobalFileStream(fileID2) as fh:
    print fh.read() #This prints "Out brief candle"

# Delete the first file from the global file-store.
job.fileStore.deleteGlobalFile(fileID)

# It is unnecessary to delete the file keyed by fileID2
# because we used the cleanup flag, which removes the file after this
# job and all its successors have run (if the file still exists)

if __name__=="__main__":
    options = Job.Runner.getDefaultOptions("./toilWorkflowRun")
    Job.Runner.startToil(Job.wrapJobFn(globalFileStoreJobFn), options)

```

The example demonstrates the global read, write and delete functionality of the file-store, using both local copies of the files and streams to read and write the files. It covers all the methods provided by the file-store interface.

What is obvious is that the file-store provides no functionality to update an existing “global” file, meaning that files are, barring deletion, immutable. Also worth noting is that there is no file system hierarchy for files in the global file store. These limitations allow us to fairly easily support different object stores and to use caching to limit the amount of network file transfer between jobs.

### 6.10.1 Staging of files into the job store

External files can be imported into or exported out of the job store prior to running a workflow when the `toil.common.Toil` context manager is used on the leader. The context manager provides methods `toil.common.Toil.importFile()`, and `toil.common.Toil.exportFile()` for this purpose. The destination and source locations of such files are described with URLs passed to the two methods. A list of the currently supported URLs can be found at `toil.jobStores.abstractJobStore.AbstractJobStore.importFile()`. To import an external file into the job store as a shared file, pass the optional `sharedFileName` parameter to that method.

If a workflow fails for any reason an imported file acts as any other file in the job store. If the workflow was configured such that it not be cleaned up on a failed run, the file will persist in the job store and needs not be staged again when the workflow is resumed.

Example:

```

from toil.common import Toil
from toil.job import Job

class HelloWorld(Job):
    def __init__(self, inputFileID):
        Job.__init__(self, memory="2G", cores=2, disk="3G")
        self.inputFileID = inputFileID

    with fileStore.readGlobalFileStream(self.inputFileID) as fi:
        with fileStore.writeGlobalFileStream() as (fo, outputFileID):
            fo.write(fi.read() + 'World!')
        return outputFileID

```

```
if __name__=="__main__":
    options = Job.Runner.getDefaultOptions("./toilWorkflowRun")
    options.logLevel = "INFO"

    with Toil(options) as toil:
        if not toil.options.restart:
            inputFileID = toil.importFile('file:///some/local/path')
            outputFileID = toil.start>HelloWorld(inputFileID))
        else:
            outputFileID = toil.restart()

    toil.exportFile(outputFileID, 'file:///some/other/local/path')
```

## 6.11 Services

It is sometimes desirable to run *services*, such as a database or server, concurrently with a workflow. The `toil.job.Job.Service` class provides a simple mechanism for spawning such a service within a Toil workflow, allowing precise specification of the start and end time of the service, and providing start and end methods to use for initialization and cleanup. The following simple, conceptual example illustrates how services work:

```
from toil.job import Job

class DemoService(Job.Service):

    def start(self, fileStore):
        # Start up a database/service here
        return "loginCredentials" # Return a value that enables another
        # process to connect to the database

    def check(self):
        # A function that if it returns False causes the service to quit
        # If it raises an exception the service is killed and an error is reported
        return True

    def stop(self, fileStore):
        # Cleanup the database here
        pass

j = Job()
s = DemoService()
loginCredentialsPromise = j.addService(s)

def dbFn(loginCredentials):
    # Use the login credentials returned from the service's start method
    # to connect to the service
    pass

j.addChildFn(dbFn, loginCredentialsPromise)

if __name__=="__main__":
    options = Job.Runner.getDefaultOptions("./toilWorkflowRun")
    Job.Runner.startToil(j, options)
```

In this example the `DemoService` starts a database in the `start` method, returning an object from the `start` method indicating how a client job would access the database. The service's `stop` method cleans up the database, while the

service's check method is polled periodically to check the service is alive.

A `DemoService` instance is added as a service of the root job `j`, with resource requirements specified. The return value from `toil.job.Job.addService()` is a promise to the return value of the service's start method. When the promise is fulfilled it will represent how to connect to the database. The promise is passed to a child job of `j`, which uses it to make a database connection. The services of a job are started before any of its successors have been run and stopped after all the successors of the job have completed successfully.

Multiple services can be created per job, all run in parallel. Additionally, services can define sub-services using `toil.job.Job.Service.addChild()`. This allows complex networks of services to be created, e.g. Apache Spark clusters, within a workflow.

## 6.12 Checkpoints

Services complicate resuming a workflow after failure, because they can create complex dependencies between jobs. For example, consider a service that provides a database that multiple jobs update. If the database service fails and loses state, it is not clear that just restarting the service will allow the workflow to be resumed, because jobs that created that state may have already finished. To get around this problem Toil supports “checkpoint” jobs, specified as the boolean keyword argument “checkpoint” to a job or wrapped function, e.g.:

```
j = Job(checkpoint=True)
```

A checkpoint job is rerun if one or more of its successors fails its retry attempts, until it itself has exhausted its retry attempts. Upon restarting a checkpoint job all its existing successors are first deleted, and then the job is rerun to define new successors. By checkpointing a job that defines a service, upon failure of the service the database and the jobs that access the service can be redefined and rerun.

To make the implementation of checkpoint jobs simple, a job can only be a checkpoint if when first defined it has no successors, i.e. it can only define successors within its run method.

## 6.13 Encapsulation

Let `A` be a root job potentially with children and follow-ons. Without an encapsulated job the simplest way to specify a job `B` which runs after `A` and all its successors is to create a parent of `A`, call it `Ap`, and then make `B` a follow-on of `Ap`. e.g.:

```
from toil.job import Job

# A is a job with children and follow-ons, for example:
A = Job()
A.addChild(Job())
A.addFollowOn(Job())

# B is a job which needs to run after A and its successors
B = Job()

# The way to do this without encapsulation is to make a
# parent of A, Ap, and make B a follow-on of Ap.
Ap = Job()
Ap.addChild(A)
Ap.addFollowOn(B)

if __name__=="__main__":
    options = Job.Runner.getDefaultOptions("./toilWorkflowRun")
    Job.Runner.startToil(Ap, options)
```

An *encapsulated job* of  $E(A)$  of  $A$  saves making  $A_p$ , instead we can write:

```
from toil.job import Job

# A
A = Job()
A.addChild(Job())
A.addFollowOn(Job())

#Encapsulate A
A = A.encapsulate()

# B is a job which needs to run after A and its successors
B = Job()

# With encapsulation A and its successor subgraph appear
# to be a single job, hence:
A.addChild(B)

if __name__=="__main__":
    options = Job.Runner.getDefaultOptions("./toilWorkflowRun")
    Job.Runner.startToil(A, options)
```

Note the call to `toil.job.Job.encapsulate()` creates the `toil.job.Job.EncapsulatedJob`.



## 7.1 Job methods

Jobs are the units of work in Toil which are composed into workflows.

**class** `toil.job.Job` (*memory=None, cores=None, disk=None, preemptable=None, cache=None, checkpoint=False*)

Class represents a unit of work in toil.

This method must be called by any overriding constructor.

### Parameters

- **memory** – the maximum number of bytes of memory the job will require to run.
- **cores** – the number of CPU cores required.
- **disk** – the amount of local disk space required by the job, expressed in bytes.
- **preemptable** – if the job can be run on a preemptable node.
- **cache** – the amount of disk (so that `cache <= disk`), expressed in bytes, for storing files from previous jobs so that they can be accessed from a local copy.
- **checkpoint** – if any of this job's successor jobs completely fails,

exhausting all their retries, remove any successor jobs and rerun this job to restart the subtree. Job must be a leaf vertex in the job graph when initially defined, see `toil.job.Job.checkNewCheckpointsAreCutVertices()`.  
 :type cores: int or string convertible by `bd2k.util.humanize.human2bytes` to an int  
 :type disk: int or string convertible by `bd2k.util.humanize.human2bytes` to an int  
 :type preemptable: boolean  
 :type cache: int or string convertible by `bd2k.util.humanize.human2bytes` to an int  
 :type memory: int or string convertible by `bd2k.util.humanize.human2bytes` to an int

**addChild** (*childJob*)

Adds `childJob` to be run as child of this job. Child jobs will be run directly after this job's `toil.job.Job.run()` method has completed.

**Parameters** `childJob` (`toil.job.Job`) –

**Returns** `childJob`

**Return type** `toil.job.Job`

**addChildFn** (*fn, \*args, \*\*kwargs*)

Adds a function as a child job.

**Parameters** **fn** – Function to be run as a child job with *\*args* and *\*\*kwargs* as arguments to this function. See `toil.job.FunctionWrappingJob` for reserved keyword arguments used to specify resource requirements.

**Returns** The new child job that wraps `fn`.

**Return type** *toil.job.FunctionWrappingJob*

**addChildJobFn** (*fn*, *\*args*, *\*\*kwargs*)

Adds a job function as a child job. See *toil.job.JobFunctionWrappingJob* for a definition of a job function.

**Parameters** **fn** – Job function to be run as a child job with *\*args* and *\*\*kwargs* as arguments to this function. See `toil.job.JobFunctionWrappingJob` for reserved keyword arguments used to specify resource requirements.

**Returns** The new child job that wraps `fn`.

**Return type** *toil.job.JobFunctionWrappingJob*

**addFollowOn** (*followOnJob*)

Adds a follow-on job, follow-on jobs will be run after the child jobs and their successors have been run.

**Parameters** **followOnJob** (*toil.job.Job*) –

**Returns** *followOnJob*

**Return type** *toil.job.Job*

**addFollowOnFn** (*fn*, *\*args*, *\*\*kwargs*)

Adds a function as a follow-on job.

**Parameters** **fn** – Function to be run as a follow-on job with *\*args* and *\*\*kwargs* as arguments to this function. See `toil.job.FunctionWrappingJob` for reserved keyword arguments used to specify resource requirements.

**Returns** The new follow-on job that wraps `fn`.

**Return type** *toil.job.FunctionWrappingJob*

**addFollowOnJobFn** (*fn*, *\*args*, *\*\*kwargs*)

Add a follow-on job function. See *toil.job.JobFunctionWrappingJob* for a definition of a job function.

**Parameters** **fn** – Job function to be run as a follow-on job with *\*args* and *\*\*kwargs* as arguments to this function. See `toil.job.JobFunctionWrappingJob` for reserved keyword arguments used to specify resource requirements.

**Returns** The new follow-on job that wraps `fn`.

**Return type** *toil.job.JobFunctionWrappingJob*

**addService** (*service*, *parentService=None*)

Add a service.

The *toil.job.Job.Service.start()* method of the service will be called after the run method has completed but before any successors are run. The service's *toil.job.Job.Service.stop()* method will be called once the successors of the job have been run.

Services allow things like databases and servers to be started and accessed by jobs in a workflow.

**Raises** *toil.job.JobException* – If service has already been made the child of a job or another service.

**Parameters**

- **service** (`toil.job.Job.Service`) – Service to add.
- **parentService** (`toil.job.Job.Service`) – Service that will be started before ‘service’ is started. Allows trees of services to be established. `parentService` must be a service of this job.

**Returns** a promise that will be replaced with the return value from `toil.job.Job.Service.start()` of service in any successor of the job.

**:rtype:** `toil.job.Promise`

**allocatePromiseFile** (*index*)

**checkJobGraphAcyclic** ()

**Raises** `toil.job.JobGraphDeadlockException` – if the connected component of jobs containing this job contains any cycles of child/followOn dependencies in the *augmented job graph* (see below). Such cycles are not allowed in valid job graphs.

A follow-on edge (A, B) between two jobs A and B is equivalent to adding a child edge to B from (1) A, (2) from each child of A, and (3) from the successors of each child of A. We call each such edge an “implied” edge. The augmented job graph is a job graph including all the implied edges.

For a job graph  $G = (V, E)$  the algorithm is  $O(|V|^2)$ . It is  $O(|V| + |E|)$  for a graph with no follow-ons. The former follow-on case could be improved!

**checkJobGraphConnected** ()

**Raises** `toil.job.JobGraphDeadlockException` – if `toil.job.Job.getRootJobs()` does not contain exactly one root job.

As execution always starts from one root job, having multiple root jobs will cause a deadlock to occur.

**checkJobGraphForDeadlocks** ()

**Raises** `toil.job.JobGraphDeadlockException` – if the job graph is cyclic, contains multiple roots or contains checkpoint jobs that are

not leaf vertices when defined (see `toil.job.Job.checkNewCheckpointsAreLeaves()`).

See `toil.job.Job.checkJobGraphConnected()`, `toil.job.Job.checkJobGraphAcyclic()` and `toil.job.Job.checkNewCheckpointsAreLeafVertices()` for more info.

**checkNewCheckpointsAreLeafVertices** ()

A checkpoint job is a job that is restarted if either it fails, or if any of its successors completely fails, exhausting their retries.

A job is a leaf if it has no successors.

A checkpoint job must be a leaf when initially added to the job graph. When its run method is invoked it can then create direct successors. This restriction is made to simplify implementation.

**Raises** `toil.job.JobGraphDeadlockException` – if there exists a job being added to the graph for which `checkpoint=True` and which is not a leaf.

**effectiveRequirements** (*config*)

Determine and validate the effective requirements for this job, substituting a missing explicit requirement with a default from the configuration.

**Return type** `Expando`

**Returns** a dictionary/object hybrid with one entry/attribute for each requirement

**encapsulate()**

Encapsulates the job, see `toil.job.EncapsulatedJob`. Convenience function for constructor of `toil.job.EncapsulatedJob`.

**Returns** an encapsulated version of this job.

**Return type** `toil.job.EncapsulatedJob`.

**getRootJobs()**

**Returns** The roots of the connected component of jobs that contains this job. A root is a job with no predecessors.

**:rtype** : set of `toil.job.Job` instances

**getTopologicalOrderingOfJobs()**

**Returns** a list of jobs such that for all pairs of indices  $i, j$  for which  $i < j$ , the job at index  $i$  can be run before the job at index  $j$ .

**Return type** list

**getUserScript()****hasChild(childJob)**

Check if `childJob` is already a child of this job.

**Parameters** **childJob** (`toil.job.Job`) –

**Returns** True if `childJob` is a child of the job, else False.

**Return type** Boolean

**run(fileStore)**

Override this function to perform work and dynamically create successor jobs.

**Parameters** **fileStore** (`toil.job.Job.FileStore`) – Used to create local and globally sharable temporary files and to send log messages to the leader process.

**Returns** The return value of the function can be passed to other jobs by means of `toil.job.Job.rv()`.

**rv(index=None)**

Creates a *promise* (`toil.job.Promise`) representing a return value of the job's run method, or, in case of a function-wrapping job, the wrapped function's return value.

**Parameters** **index** (`int` | `None`) – If `None` the complete return value will be used, otherwise an index to select an individual item from the return value in which case the return value must be of a type that implements the `__getitem__` magic method, e.g. dict, list or tuple.

**Returns** A promise representing the return value of this jobs `toil.job.Job.run()` method.

**Return type** `toil.job.Promise`

**static wrapFn(fn, \*args, \*\*kwargs)**

Makes a Job out of a function. Convenience function for constructor of `toil.job.FunctionWrappingJob`.

**Parameters** **fn** – Function to be run with `*args` and `**kwargs` as arguments. See `toil.job.JobFunctionWrappingJob` for reserved keyword arguments used to specify resource requirements.

**Returns** The new function that wraps `fn`.

**Return type** `toil.job.FunctionWrappingJob`

**static wrapJobFn** (*fn*, *\*args*, *\*\*kwargs*)

Makes a Job out of a job function. Convenience function for constructor of *toil.job.JobFunctionWrappingJob*.

**Parameters** *fn* – Job function to be run with *\*args* and *\*\*kwargs* as arguments. See *toil.job.JobFunctionWrappingJob* for reserved keyword arguments used to specify resource requirements.

**Returns** The new job function that wraps *fn*.

**Return type** *toil.job.JobFunctionWrappingJob*

## 7.2 Job.FileStore

The FileStore is an abstraction of a Toil run’s shared storage.

**class** *Job.FileStore* (*jobStore*, *jobWrapper*, *localTempDir*, *inputBlockFn*)

Class used to manage temporary files, read and write files from the job store and log messages, passed as argument to the *toil.job.Job.run()* method.

This constructor should not be called by the user, FileStore instances are only provided as arguments to the run function.

### Parameters

- **jobStore** (*toil.jobStores.abstractJobStore.JobStore*) – The job store for the workflow.
- **jobWrapper** (*toil.jobWrapper.JobWrapper*) – The jobWrapper for the job.
- **localTempDir** (*string*) – A temporary directory in which local temporary files will be placed.
- **inputBlockFn** (*method*) – A function which blocks and which is called before the fileStore completes atomically updating the jobs files in the job store.

**asyncWrite** ()

**deleteGlobalFile** (*fileStoreID*)

Deletes a global file with the given job store ID.

To ensure that the job can be restarted if necessary, the delete will not happen until after the job’s run method has completed.

**Parameters** *fileStoreID* – the job store ID of the file to be deleted.

**exportFile** (*jobStoreFileID*, *dstUrl*)

**getLocalTempDir** ()

Get a new local temporary directory in which to write files that persist for the duration of the job.

**Returns** The absolute path to a new local temporary directory. This directory will exist for the duration of the job only, and is guaranteed to be deleted once the job terminates, removing all files it contains recursively.

**Return type** *string*

**getLocalTempFile** ()

Get a new local temporary file that will persist for the duration of the job.

**Returns** The absolute path to a local temporary file. This file will exist for the duration of the job only, and is guaranteed to be deleted once the job terminates.

**Return type** string

**getLocalTempFileName** ()

Get a valid name for a new local file. Do it in a really stupid way by creating and then deleting a temp file (haha). :return: Path to valid file

**importFile** (*srcUrl*, *sharedFileName=None*)

**logToMaster** (*text*, *level=20*)

Send a logging message to the leader. The message will also be logged by the worker at the same level.

**Parameters**

- **text** – The string to log.
- **level** (*int*) – The logging level.

**open** (*\*args*, *\*\*kws*)

This is a dummy context manager that has a true purpose in Job.CachedFileStore where the `__enter__` and `__exit__` methods carry out cache eviction, and cache cleanup operations. :param job: :return:

**readGlobalFile** (*fileStoreID*, *userPath=None*, *cache=True*, *mutable=None*)

Get a copy of a file in the job store.

**Parameters**

- **userPath** (*string*) – a path to the name of file to which the global file will be copied or hard-linked (see below).
- **cache** (*boolean*) – If True will use caching (see below). Caching will attempt to keep copies of files between sequences of jobs run on the same worker.
- **mutable** (*boolean*) – If True, the file path returned points to a file that is

modifiable by the user. The value defaults to the False unless backwards compatibility was requested.

If `cache=True` and `userPath` is either: (1) a file path contained within a directory or, recursively, a subdirectory of a temporary directory returned by `Job.FileStore.getLocalTempDir()`, or (2) a file path returned by `Job.FileStore.getLocalTempFile()` then the file will be cached and returned file will be read only (have permissions 444).

If `userPath` is specified and the file is already cached, the `userPath` file will be a hard link to the actual location, else it will be an actual copy of the file.

If the `cache=False` or `userPath` is not either of the above the file will not be cached and will have default permissions. Note, if the file is already cached this will result in two copies of the file on the system.

**Returns** an absolute path to a local, temporary copy of the file keyed by `fileStoreID`.

:rtype : string

**readGlobalFileStream** (*fileStoreID*)

Similar to `readGlobalFile`, but allows a stream to be read from the job store.

**Returns** a context manager yielding a file handle which can be read from. The yielded file handle does not need to and should not be closed explicitly.

**writeGlobalFile** (*localFileName*, *cleanup=False*)

Takes a file (as a path) and uploads it to the job store.

If the local file is a file returned by `toil.job.Job.FileStore.getLocalTempFile()` or is in a directory, or, recursively, a subdirectory, returned by `toil.job.Job.FileStore.getLocalTempDir()` then the write is asynchronous, so further modifications during execution to the file pointed by `localFileName` will result in undetermined behavior. Otherwise, the method will block until the file is written to the file store.

**Parameters**

- **localFileName** (*string*) – The path to the local file to upload.
- **cleanup** (*Boolean*) – if True then the copy of the global file will be deleted once the job and all its successors have completed running. If not the global file must be deleted manually.

**Returns** an ID that can be used to retrieve the file.

**writeGlobalFileStream** (*cleanup=False*)

Similar to writeGlobalFile, but allows the writing of a stream to the job store.

**Parameters** **cleanup** (*Boolean*) – is as in `toil.job.Job.FileStore.writeGlobalFile()`.

**Returns** a context manager yielding a tuple of 1) a file handle which can be written to and 2) the ID of the resulting file in the job store. The yielded file handle does not need to and should not be closed explicitly.

## 7.3 Job.Runner

The Runner contains the methods needed to configure and start a Toil run.

**class** `Job.Runner`

Used to setup and run Toil workflow.

**static** **addToilOptions** (*parser*)

Adds the default toil options to an `optparse` or `argparse` parser object.

**Parameters** **parser** (*optparse.OptionParser or argparse.ArgumentParser*)  
– Options object to add toil options to.

**static** **getDefaultArgumentParser** ()

Get argument parser with added toil workflow options.

**Returns** The argument parser used by a toil workflow with added Toil options.

**Return type** `argparse.ArgumentParser`

**static** **getDefaultOptions** (*jobStore*)

Get default options for a toil workflow.

**Parameters** **jobStore** (*string*) – A string describing the jobStore for the workflow.

**Returns** The options used by a toil workflow.

**Return type** `argparse.ArgumentParser` values object

**static** **startToil** (*job, options*)

Deprecated by `toil.common.Toil.run`. Runs the toil workflow using the given options (see `Job.Runner.getDefaultOptions` and `Job.Runner.addToilOptions`) starting with this job. :param `toil.job.Job` job: root job of the workflow :raises: `toil.leader.FailedJobsException` if at the end of function their remain failed jobs. :return: The return value of the root job's run function. :rtype: Any

## 7.4 Toil

The `Toil` class provides for a more general way to configure and start a Toil run.

**class** `toil.common.Toil` (*options*)

A context manager that represents a Toil workflow, specifically the batch system, job store, and its configuration.

Initialize a Toil object from the given options. Note that this is very light-weight and that the bulk of the work is done when the context is entered.

**Parameters** `options` (*argparse.Namespace*) – command line options specified by the user

**static** `createBatchSystem` (*config, jobStore=None, userScript=None*)

Creates an instance of the batch system specified in the given config. If a job store and a user script are given then the user script can be hot deployed into the workflow.

**Parameters**

- **config** (*toil.common.Config*) – the current configuration
- **jobStore** (*jobStores.abstractJobStore.AbstractJobStore*) – an instance of a jobStore
- **userScript** (*ModuleDescriptor*) – a user supplied script to use for hot development

**Returns** an instance of a concrete subclass of `AbstractBatchSystem`

**Return type** *batchSystems.abstractBatchSystem.AbstractBatchSystem*

**exportFile** (*jobStoreFileID, dstUrl*)

**static** `getWorkflowDir` (*workflowID, configWorkDir=None*)

Returns a path to the directory where worker directories and the cache will be located for this workflow.

**Parameters**

- **workflowID** (*str*) – Unique identifier for the workflow
- **configWorkDir** (*str*) – Value passed to the program using the `–workDir` flag

**Returns** Path to the workflow directory

**Return type** *str*

**importFile** (*srcUrl, sharedFileName=None*)

**static** `loadOrCreateJobStore` (*jobStoreString, config=None*)

Loads an existing jobStore if it already exists. Otherwise a new instance of a jobStore is created and returned.

**Parameters**

- **jobStoreString** (*str*) – see exception message below
- **config** (*toil.common.Config*) – see `AbstractJobStore.__init__`

**Returns** an instance of a concrete subclass of `AbstractJobStore`

**Return type** *toil.jobStores.abstractJobStore.AbstractJobStore*

**restart** ()

Restarts a workflow that has been interrupted. This method should be called if and only if a workflow has previously been started and has not finished.

**Returns** The root job's return value

**start** (*rootJob*)

Invoke a Toil workflow with the given job as the root for an initial run. This method must be called in the body of a `with Toil(...)` as `toil:` statement. This method should not be called more than once for a workflow that has not finished.



**Parameters** `rootJob` (`toil.job.Job`) – The root job of the workflow

**Returns** The root job’s return value

## 7.5 Job.Service

The Service class allows databases and servers to be spawned within a Toil workflow.

**class** `Job.Service` (*memory=None, cores=None, disk=None, preemptable=None*)

Abstract class used to define the interface to a service.

Memory, core and disk requirements are specified identically to as in `toil.job.Job.__init__()`.

**check** ()

Checks the service is still running.

**Raises** `RuntimeError` – If the service failed, this will cause the service job to be labeled failed.

**Returns** True if the service is still running, else False. If False then the service job will be terminated,

and considered a success. Important point: if the service job exits due to a failure, it should raise a `RuntimeError`, not return False!

**start** (*fileStore*)

Start the service.

**Parameters** `fileStore` (`toil.job.Job.FileStore`) – A `fileStore` object to create temporary files with.

**Returns** An object describing how to access the service. The object must be pickleable and will be used by jobs to access the service (see `toil.job.Job.addService()`).

**stop** (*fileStore*)

Stops the service.

**Parameters** `fileStore` (`toil.job.Job.FileStore`) – A `fileStore` object to create temporary files with.

Function can block until complete.

## 7.6 FunctionWrappingJob

The subclass of Job for wrapping user functions.

**class** `toil.job.FunctionWrappingJob` (*userFunction, \*args, \*\*kwargs*)

Job used to wrap a function. In its run method the wrapped function is called.

**Parameters** `userFunction` – The function to wrap. The `userFunction` will be called with the `*args` and `**kwargs` as arguments.

The keywords “memory”, “cores”, “disk”, “cache” are reserved keyword arguments that if specified will be used to determine the resources for the job, as `toil.job.Job.__init__()`. If they are keyword arguments to the function they will be extracted from the function definition, but may be overridden by the user (as you would expect).

**getUserScript** ()

**run** (*fileStore*)

## 7.7 JobFunctionWrappingJob

The subclass of `FunctionWrappingJob` for wrapping user job functions.

**class** `toil.job.JobFunctionWrappingJob` (*userFunction*, \*args, \*\*kwargs)

A job function is a function whose first argument is a `job.Job` instance that is the wrapping job for the function. This can be used to add successor jobs for the function and perform all the functions the `job.Job` class provides.

To enable the job function to get access to the `toil.job.Job.FileStore` instance (see `toil.job.Job.Run()`), it is made a variable of the wrapping job called `fileStore`.

**Parameters** `userFunction` – The function to wrap. The `userFunction` will be called with the `*args` and `**kwargs` as arguments.

The keywords “memory”, “cores”, “disk”, “cache” are reserved keyword arguments that if specified will be used to determine the resources for the job, as `toil.job.Job.__init__()`. If they are keyword arguments to the function they will be extracted from the function definition, but may be overridden by the user (as you would expect).

**run** (*fileStore*)

## 7.8 EncapsulatedJob

The subclass of `Job` for *encapsulating* a job, allowing a subgraph of jobs to be treated as a single job.

**class** `toil.job.EncapsulatedJob` (*job*)

A convenience `Job` class used to make a job subgraph appear to be a single job.

Let A be the root job of a job subgraph and B be another job we’d like to run after A and all its successors have completed, for this use `encapsulate`:

```
A, B = A(), B() #Job A and subgraph, Job B
A' = A.encapsulate()
A'.addChild(B) #B will run after A and all its successors have
# completed, A and its subgraph of successors in effect appear
# to be just one job.
```

The return value of an encapsulated job (as accessed by the `toil.job.Job.rv()` function) is the return value of the root job, e.g. `A().encapsulate().rv()` and `A().rv()` will resolve to the same value after A or `A.encapsulate()` has been run.

**Parameters** `job` (`toil.job.Job`) – the job to encapsulate.

**addChild** (*childJob*)

**addFollowOn** (*followOnJob*)

**addService** (*service*)

**rv** (*index=None*)

## 7.9 Promise

The class used to reference return values of jobs/services not yet run/started.

**class** `toil.job.Promise(job, index)`

References a return value from a `toil.job.Job.run()` or `toil.job.Job.Service.start()` method as a *promise* before the method itself is run.

Let T be a job. Instances of `Promise` (termed a *promise*) are returned by `T.rv()`, which is used to reference the return value of T's run function. When the promise is passed to the constructor (or as an argument to a wrapped function) of a different, successor job the promise will be replaced by the actual referenced return value. This mechanism allows a return values from one job's run method to be input argument to job before the former job's run function has been executed.

**Parameters** `job` (`Job`) – the job whose return value this promise references

**filesToDelete** = `set()`

A set of IDs of files containing promised values when we know we won't need them anymore

**class** `toil.job.PromisedRequirement(valueOrCallable, *args)`

Class for dynamically allocating job function resource requirements involving `toil.job.Promise` instances.

Use when resource requirements depend on the return value of a parent function. `PromisedRequirements` can be modified by passing a function that takes the `Promise` as input.

For example, let f, g, and h be functions. Then a Toil workflow can be defined as follows:

```
A = Job.wrapFn(f)
B = A.addChildFn(g, cores=PromisedRequirement(A.rv()))
C = B.addChildFn(h, cores=PromisedRequirement(lambda x: 2*x, B.rv()))
```

**Parameters**

- **valueOrCallable** – A single `Promise` instance or a function that takes `*args` as input parameters.
- **\*args** (`int | Promise`) – variable length argument list

**static** `convertPromises(kwargs)`

Returns True if reserved resource keyword is a `Promise` or `PromisedRequirement` instance. Converts `Promise` instance to `PromisedRequirement`.

**Parameters** `kwargs` – function keyword arguments

**Returns** `bool`

**getValue()**

Returns `PromisedRequirement` value

## 7.10 Exceptions

Toil specific exceptions.

**class** `toil.job.JobException(message)`

General job exception.

**class** `toil.job.JobGraphDeadlockException(string)`

An exception raised in the event that a workflow contains an unresolvable dependency, such as a cycle. See `toil.job.Job.checkJobGraphForDeadlocks()`.



---

## Toil architecture

---

The following diagram layouts out the software architecture of Toil.

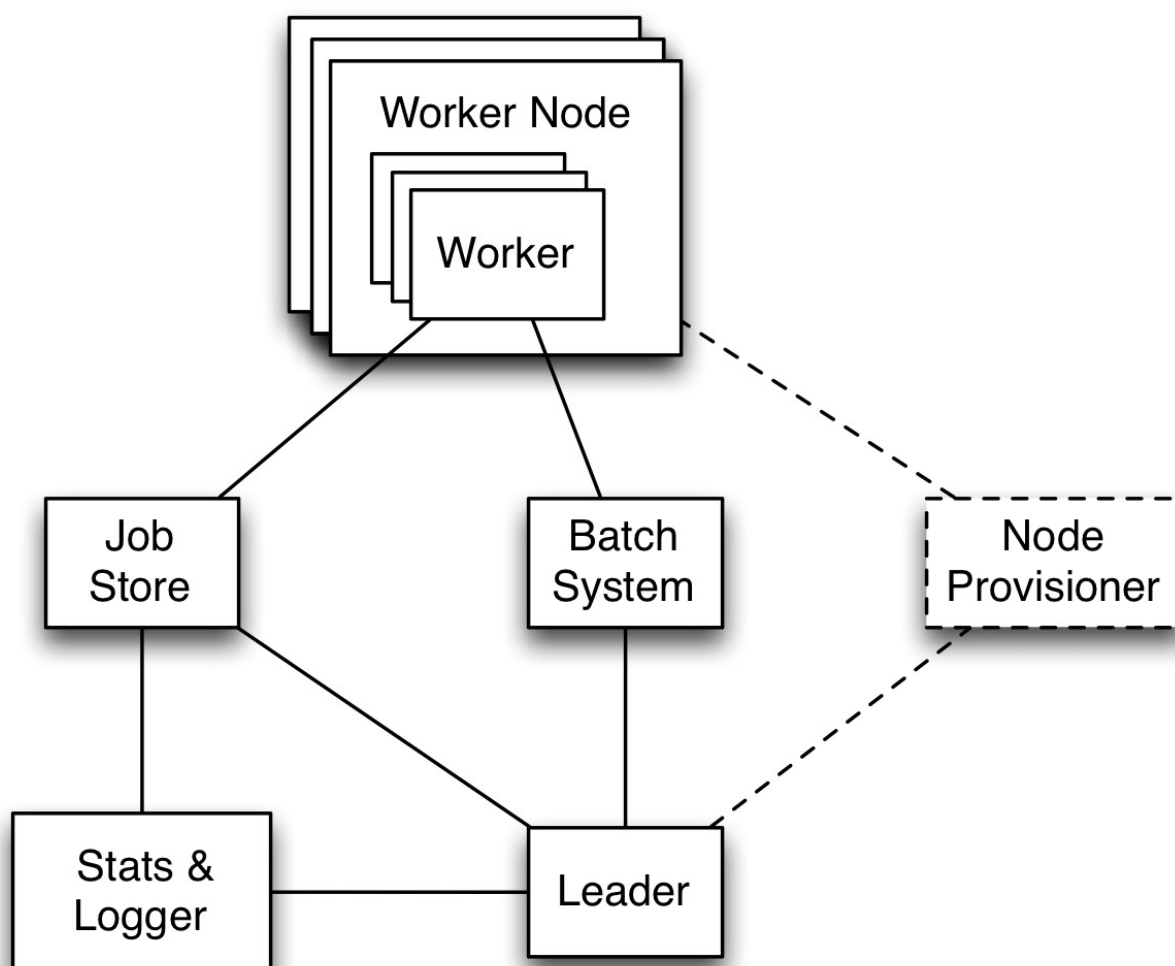


Fig. 8.1: Figure 1: The basic components of the toil architecture. Note the node provisioning is coming soon.

These components are described below:

- **the leader:** The leader is responsible for deciding which jobs should be run. To do this it traverses the job graph. Currently this is a single threaded process, but we make aggressive steps to prevent it becoming a bottleneck (see [Read-only Leader](#) described below).
- **the job-store:** Handles all files shared between the components. Files in the job-store are the means by which the state of the workflow is maintained. Each job is backed by a file in the job store, and atomic updates to this state are used to ensure the workflow can always be resumed upon failure. The job-store can also store all user files, allowing them to be shared between jobs. The job-store is defined by the abstract class `toil.jobStores.AbstractJobStore`. Multiple implementations of this class allow Toil to support different back-end file stores, e.g.: S3, network file systems, Azure file store, etc.
- **workers:** The workers are temporary processes responsible for running jobs, one at a time per worker. Each worker process is invoked with a job argument that it is responsible for running. The worker monitors this job and reports back success or failure to the leader by editing the job's state in the file-store. If the job defines successor jobs the worker may choose to immediately run them (see [Job Chaining](#) below).
- **the batch-system:** Responsible for scheduling the jobs given to it by the leader, creating a worker command for each job. The batch-system is defined by the abstract class `toil.batchSystems.AbstractBatchSystem`. Toil uses multiple existing batch systems to schedule jobs, including Apache Mesos, GridEngine and a multi-process single node implementation that allows workflows to be run without any of these frameworks. Toil can therefore fairly easily be made to run a workflow using an existing cluster.
- **the node provisioner:** Creates worker nodes in which the batch system schedules workers. This is currently being developed. It is defined by the abstract class `toil.provisioners.AbstractProvisioner`.
- **the statistics and logging monitor:** Monitors logging and statistics produced by the workers and reports them. Uses the job-store to gather this information.

## 8.1 Optimizations

Toil implements lots of optimizations designed for scalability. Here we detail some of the key optimizations.

### 8.1.1 Read-only leader

The leader process is currently implemented as a single thread. Most of the leader's tasks revolve around processing the state of jobs, each stored as a file within the job-store. To minimise the load on this thread, each worker does as much work as possible to manage the state of the job it is running. As a result, with a couple of minor exceptions, the leader process never needs to write or update the state of a job within the job-store. For example, when a job is complete and has no further successors the responsible worker deletes the job from the job-store, marking it complete. The leader then only has to check for the existence of the file when it receives a signal from the batch-system to know that the job is complete. This off-loading of state management is orthogonal to future parallelization of the leader.

### 8.1.2 Job chaining

The scheduling of successor jobs is partially managed by the worker, reducing the number of individual jobs the leader needs to process. Currently this is very simple: if there is a single next successor job to run and its resources fit within the resources of the current job and closely match the resources of the current job then the job is run immediately on the worker without returning to the leader. Further extensions of this strategy are possible, but for many workflows

which define a series of serial successors (e.g. map sequencing reads, post-process mapped reads, etc.) this pattern is very effective at reducing leader workload.

### 8.1.3 Preemptable node support

Critical to running at large-scale is dealing with intermittent node failures. Toil is therefore designed to always be resumable providing the job-store does not become corrupt. This robustness allows Toil to run on preemptible nodes, which are only available when others are not willing to pay more to use them. Designing workflows that divide into many short individual jobs that can use preemptable nodes allows for workflows to be efficiently scheduled and executed.

### 8.1.4 Caching

Running bioinformatic pipelines often require the passing of large datasets between jobs. Toil caches the results from jobs such that child jobs running on the same node can directly use the same file objects, thereby eliminating the need for an intermediary transfer to the job store. Caching also reduces the burden on the local disks, because multiple jobs can share a single file. The resulting drop in I/O allows pipelines to run faster, and, by the sharing of files, allows users to run more jobs in parallel by reducing overall disk requirements.

To demonstrate the efficiency of caching, we ran an experimental internal pipeline on 3 samples from the TCGA Lung Squamous Carcinoma (LUSC) dataset. The pipeline takes the tumor and normal exome fastqs, and the tumor rna fastq and input, and predicts MHC presented neopeptides in the patient that are potential targets for T-cell based immunotherapies. The pipeline was run individually on the samples on c3.8xlarge machines on AWS (60GB RAM, 600GB SSD storage, 32 cores). The pipeline aligns the data to hg19-based references, predicts MHC haplotypes using PHLAT, calls mutations using 2 callers (MuTect and RADIA) and annotates them using SnpEff, then predicts MHC:peptide binding using the IEDB suite of tools before running an in-house rank boosting algorithm on the final calls.

To optimize time taken, The pipeline is written such that mutations are called on a per-chromosome basis from the whole-exome bams and are merged into a complete vcf. Running mutect in parallel on whole exome bams requires each mutect job to download the complete Tumor and Normal Bams to their working directories – An operation that quickly fills the disk and limits the parallelizability of jobs. The script was run in Toil, with and without caching, and Figure 2 shows that the workflow finishes faster in the cached case while using less disk on average than the uncached run. We believe that benefits of caching arising from file transfers will be much higher on magnetic disk-based storage systems as compared to the SSD systems we tested this on.

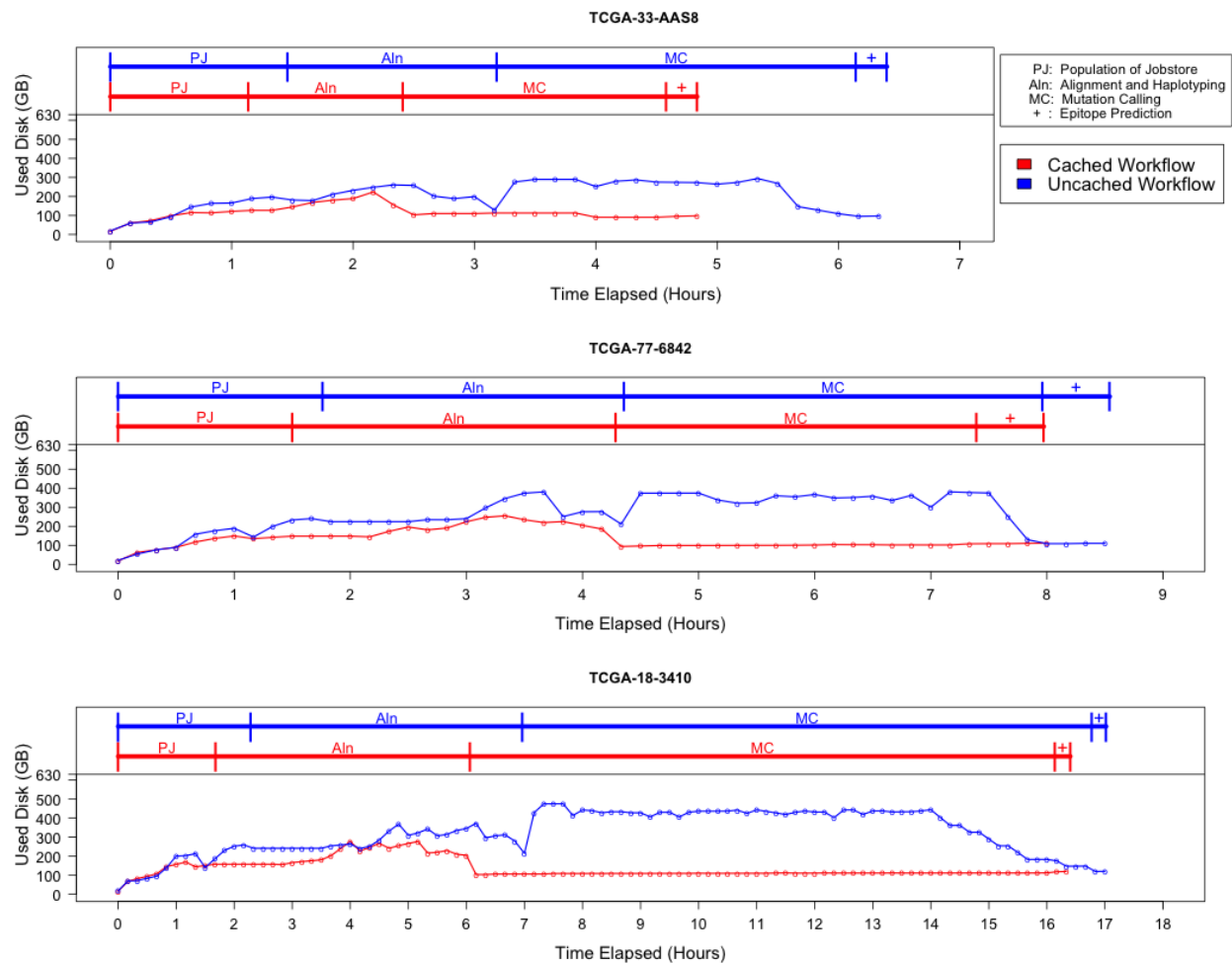


Fig. 8.2: Figure 2: Efficiency gain from caching. The lower half of each plot describes the disk used by the pipeline recorded every 10 minutes over the duration of the pipeline, and the upper half shows the corresponding stage of the pipeline that is being processed. Since jobs requesting the same file shared the same inode, the effective load on the disk is considerably lower than in the uncached case where every job downloads a personal copy of every file it needs. We see that in all cases, the uncached run uses almost 300-400GB more than the uncached run in the resource heavy mutation calling step. We also see a benefit in terms of wall time for each stage since we eliminate the time taken for file transfers.



---

## The batch system interface

---

The batch system interface is used by Toil to abstract over different ways of running batches of jobs, for example GridEngine, Mesos, Parasol and a single node. The `toil.batchSystems.abstractBatchSystem.AbstractBatchSystem` API is implemented to run jobs using a given job management system, e.g. Mesos.

**class** `toil.batchSystems.abstractBatchSystem.AbstractBatchSystem`

An abstract (as far as Python currently allows) base class to represent the interface the batch system must provide to Toil.

**getIssuedBatchJobIDs** ()

Gets all currently issued jobs

**Returns** A list of jobs (as jobIDs) currently issued (may be running, or may be waiting to be run). Despite the result being a list, the ordering should not be depended upon.

**Return type** `list[str]`

**classmethod** `getRescueBatchJobFrequency` ()

Gets the period of time to wait (floating point, in seconds) between checking for missing/overlong jobs.

**getRunningBatchJobIDs** ()

Gets a map of jobs as jobIDs that are currently running (not just waiting) and how long they have been running, in seconds.

**Returns** dictionary with currently running jobID keys and how many seconds they have been running as the value

**Return type** `dict[str,float]`

**getUpdatedBatchJob** (*maxWait*)

Returns a job that has updated its status.

**Parameters** **maxWait** (*float*) – the number of seconds to block, waiting for a result

**Return type** `(str, int)|None`

**Returns** If a result is available, returns a tuple (jobID, exitValue, wallTime). Otherwise it returns None. wallTime is the number of seconds (a float) in wall-clock time the job ran for or None if this batch system does not support tracking wall time. Returns None for jobs that were killed.

**issueBatchJob** (*command, memory, cores, disk, preemptable*)

Issues a job with the specified command to the batch system and returns a unique jobID.

**Parameters**

- **command** (*str*) – the string to run as a command,

- **memory** (*int*) – int giving the number of bytes of memory the job needs to run
- **cores** (*float*) – the number of cores needed for the job
- **disk** (*int*) – int giving the number of bytes of disk space the job needs to run
- **preemptable** (*boolean*) – True if the job can be run on a preemptable node

**Returns** a unique jobID that can be used to reference the newly issued job

**Return type** int

**killBatchJobs** (*jobIDs*)

Kills the given job IDs.

**Parameters** **jobIDs** (*list[int]*) – list of IDs of jobs to kill

**setEnv** (*name, value=None*)

Set an environment variable for the worker process before it is launched. The worker process will typically inherit the environment of the machine it is running on but this method makes it possible to override specific variables in that inherited environment before the worker is launched. Note that this mechanism is different to the one used by the worker internally to set up the environment of a job. A call to this method affects all jobs issued after this method returns. Note to implementors: This means that you would typically need to copy the variables before enqueueing a job.

If no value is provided it will be looked up from the current environment.

**shutdown** ()

Called at the completion of a toil invocation. Should cleanly terminate all worker threads.

**classmethod supportsHotDeployment** ()

Whether this batch system supports hot deployment of the user script and toil itself. If it does, the `__init__` method will have to accept two optional parameters in addition to the declared ones: `userScript` and `toilDistribution`. Both will be instances of `toil.common.HotDeployedResource` that represent the user script and a source tarball ( `sdist`) of toil respectively.

**Return type** bool

**classmethod supportsWorkerCleanup** ()

Indicates whether this batch system invokes `workerCleanup()` after the last job for a particular workflow invocation finishes. Note that the term *worker* refers to an entire node, not just a worker process. A worker process may run more than one job sequentially, and more than one concurrent worker process may exist on a worker node, for the same workflow. The batch system is said to *shut down* after the last worker process terminates.

**Return type** bool

---

## The job store interface

---

The job store interface is an abstraction layer that hides the specific details of file storage, for example standard file systems, S3, etc. The `toil.jobStores.abstractJobStore.AbstractJobStore` API is implemented to support a given file store, e.g. S3. Implement this API to support a new file store.

**class** `toil.jobStores.abstractJobStore.AbstractJobStore` (*config=None*)

Represents the physical storage for the jobs and associated files in a toil.

**Parameters** `config` (`toil.common.Config`) – If `config` is not `None` then the given configuration object will be written to the shared file “config.pickle” which can later be retrieved using the `readSharedFileStream`. See `writeConfigToStore`. If this file already exists it will be overwritten. If `config` is `None`, the shared file “config.pickle” is assumed to exist and is retrieved. See `loadConfigFromStore`.

**clean** (*jobCache=None*)

Function to cleanup the state of a job store after a restart. Fixes jobs that might have been partially updated. Resets the try counts and removes jobs that are not successors of the current root job.

**Parameters** `jobCache` (`dict[str, toil.jobWrapper.JobWrapper]`) – if a value it must be a dict from job ID keys to `JobWrapper` object values. Jobs will be loaded from the cache (which can be downloaded from the job store in a batch) instead of piecemeal when recursed into.

**config**

The Toil configuration associated with this job store.

**Return type** `toil.common.Config`

**create** (*command, memory, cores, disk, preemptable, predecessorNumber=0*)

Creates a `jobWrapper` with specified resources and command, adds it to the job store and returns it.

**Parameters**

- **command** (*str*) – the shell command that will be executed when the job is being run
- **memory** (*int*) – the amount of RAM in bytes needed to run the job
- **cores** (*float*) – the number of cores needed to run the job
- **disk** (*int*) – the amount of disk in bytes needed to run the job
- **preemptable** (*bool*) – whether the job can be run on a preemptable node
- **predecessorNumber** (*int*) – argument to the job constructor. Specifies the number of other `jobWrappers` that specify this job in their stack

**Returns** the newly created `jobWrapper` object

**Return type** `toil.jobWrapper.JobWrapper`

**createRootJob** (*\*args, \*\*kwargs*)

Create a new job and set it as the root job in this job store

**:rtype** : `toil.jobWrapper.JobWrapper`

**delete** (*jobStoreID*)

Removes from store atomically, can not then subsequently call `load()`, `write()`, `update()`, etc. with the job.

This operation is idempotent, i.e. deleting a job twice or deleting a non-existent job will succeed silently.

**Parameters** **jobStoreID** (*str*) – the ID of the job to delete from this job store

**deleteFile** (*jobStoreFileID*)

Deletes the file with the given ID from this job store. This operation is idempotent, i.e. deleting a file twice or deleting a non-existent file will succeed silently.

**Parameters** **jobStoreFileID** (*str*) – ID of the file to delete

**deleteJobStore** ()

Removes the job store from the disk/store. Careful!

**exists** (*jobStoreID*)

Indicates whether the job with the specified `jobStoreID` exists in the job store

**Return type** `bool`

**exportFile** (*jobStoreFileID, dstUrl*)

Exports file to destination pointed at by the destination URL.

Refer to `AbstractJobStore.importFile` documentation for currently supported URL schemes.

Note that the helper method `_exportFile` is used to read from the source and write to destination. To implement any optimizations that circumvent this, the `_exportFile` method should be overridden by subclasses of `AbstractJobStore`.

**Parameters**

- **jobStoreFileID** (*str*) – The id of the file in the job store that should be exported.
- **dstUrl** (*str*) – URL that points to a file or object in the storage mechanism of a supported URL scheme e.g. a blob in an Azure Blob Storage container.

**fileExists** (*jobStoreFileID*)

Determine whether a file exists in this job store.

**Parameters** **jobStoreFileID** (*str*) – an ID referencing the file to be checked

**Return type** `bool`

**getEmptyFileStoreID** (*jobStoreID=None*)

Creates an empty file in the job store and returns its ID. Call to `fileExists(getEmptyFileStoreID(jobStoreID))` will return `True`.

**Parameters** **jobStoreID** (*str*) – the id of a job, or `None`. If specified, the file will be associated with that job and when `jobStore.delete(job)` is called a best effort attempt is made to delete all files written with the given `job.jobStoreID`

**Returns** a `jobStoreFileID` that references the newly created file and can be used to reference the file in the future.

**Return type** `str`

**getEnv()**

Returns a dictionary of environment variables that this job store requires to be set in order to function properly on a worker.

**Return type** dict[str,str]

**getPublicUrl(fileName)**

Returns a publicly accessible URL to the given file in the job store. The returned URL may expire as early as 1h after its been returned. Throw an exception if the file does not exist.

**Parameters** **fileName** (*str*) – the jobStoreFileID of the file to generate a URL for

**Raises** **NoSuchFileException** – if the specified file does not exist in this job store

**Return type** str

**getSharedPublicUrl(sharedFileName)**

Differs from `getPublicUrl()` in that this method is for generating URLs for shared files written by `writeSharedFileStream()`.

Returns a publicly accessible URL to the given file in the job store. The returned URL starts with 'http:', 'https:' or 'file:'. The returned URL may expire as early as 1h after its been returned. Throw an exception if the file does not exist.

**Parameters** **sharedFileName** (*str*) – The name of the shared file to generate a publically accessible url for.

**Raises** **NoSuchFileException** – raised if the specified file does not exist in the store

**Return type** str

**importFile(srcUrl, sharedFileName=None)**

Imports the file at the given URL into job store. The jobStoreFileId of the new file is returned. If a shared file name is given, the file will be imported as a shared file and None is returned.

Note that the helper method `_importFile` is used to read from the source and write to destination (which is the current job store in this case). To implement any optimizations that circumvent this, the `_importFile` method should be overridden by subclasses of `AbstractJobStore`.

**Currently supported schemes are:**

- **'s3' for objects in Amazon S3** e.g. `s3://bucket/key`
- **'wasb' for blobs in Azure Blob Storage** e.g. `wasb://container/blob`
- **'file' for local files** e.g. `file:///local/file/path`
- **'http'** e.g. `http://someurl.com/path`

**Parameters**

- **srcUrl** (*str*) – URL that points to a file or object in the storage mechanism of a supported URL scheme e.g. a blob in an Azure Blob Storage container.
- **sharedFileName** (*str*) – Optional name to assign to the imported file within the job store

:return The jobStoreFileId of the imported file or None if sharedFileName was given :rtype: str|None

**jobs()**

Best effort attempt to return iterator on all jobs in the store. The iterator may not return all jobs and may also contain orphaned jobs that have already finished successfully and should not be rerun. To guarantee you get any and all jobs that can be run instead construct a more expensive `ToilState` object

**Returns** Returns iterator on jobs in the store. The iterator may or may not contain all jobs and may contain invalid jobs

**Return type** `Iterator[toil.jobWrapper.JobWrapper]`

**load** (*jobStoreID*)

Loads the job referenced by the given ID and returns it.

**Parameters** **jobStoreID** (*str*) – the ID of the job to load

**Raises** **NoSuchJobException** – if there is no job with the given ID

**Return type** `toil.jobWrapper.JobWrapper`

**loadRootJob** ()

Loads the root job in the current job store.

**Raises** `toil.job.JobException` – If no root job is set or if the root job doesn't exist in this job store

**Returns** The root job.

**Return type** `toil.jobWrapper.JobWrapper`

**publicUrlExpiration** = `datetime.timedelta(365)`

**readFile** (*jobStoreFileID*, *localFilePath*)

Copies the file referenced by *jobStoreFileID* to the given local file path. The version will be consistent with the last copy of the file written/updated.

The file at the given local path may not be modified after this method returns!

**Parameters**

- **jobStoreFileID** (*str*) – ID of the file to be copied
- **localFilePath** (*str*) – the local path indicating where to place the contents of the given file in the job store

**readFileStream** (*\*args*, *\*\*kws*)

Similar to `readFile`, but returns a context manager yielding a file handle which can be read from. The yielded file handle does not need to and should not be closed explicitly.

**Parameters** **jobStoreFileID** (*str*) – ID of the file to get a readable file handle for

**readSharedFileStream** (*\*args*, *\*\*kws*)

Returns a context manager yielding a readable file handle to the global file referenced by the given name.

**Parameters** **sharedFileName** (*str*) – A file name matching `AbstractJobStore.fileNameRegex`, unique within this job store

**readStatsAndLogging** (*callback*, *readAll=False*)

Reads stats/logging strings accumulated by the `writeStatsAndLogging()` method. For each stats/logging string this method calls the given callback function with an open, readable file handle from which the stats string can be read. Returns the number of stats/logging strings processed. Each stats/logging string is only processed once unless the `readAll` parameter is set, in which case the given callback will be invoked for all existing stats/logging strings, including the ones from a previous invocation of this method.

**Parameters**

- **callback** (*Callable*) – a function to be applied to each of the stats file handles found
- **readAll** (*bool*) – a boolean indicating whether to read the already processed stats files in addition to the unread stats files

**Raises** **ConcurrentFileModificationException** – if the file was modified concurrently during an invocation of this method

**Returns** the number of stats files processed

**Return type** int

**rootJobStoreIDFileName** = 'rootJobStoreID'

**setRootJob** (*rootJobStoreID*)

Set the root job of the workflow backed by this job store

**Parameters** **rootJobStoreID** (*str*) – The ID of the job to set as root

**sharedFileNameRegex** = <\_sre.SRE\_Pattern object>

**update** (*job*)

Persists the job in this store atomically.

**Parameters** **job** (*toil.jobWrapper.JobWrapper*) – the job to write to this job store

**updateFile** (*jobStoreFileID, localFilePath*)

Replaces the existing version of a file in the job store. Throws an exception if the file does not exist.

**Parameters**

- **jobStoreFileID** (*str*) – the ID of the file in the job store to be updated
- **localFilePath** (*str*) – the local path to a file that will overwrite the current version in the job store

**Raises**

- **ConcurrentFileModificationException** – if the file was modified concurrently during an invocation of this method
- **NoSuchFileException** – if the specified file does not exist

**updateFileStream** (*jobStoreFileID*)

Replaces the existing version of a file in the job store. Similar to `writeFile`, but returns a context manager yielding a file handle which can be written to. The yielded file handle does not need to and should not be closed explicitly.

**Parameters** **jobStoreFileID** (*str*) – the ID of the file in the job store to be updated

**Raises**

- **ConcurrentFileModificationException** – if the file was modified concurrently during an invocation of this method
- **NoSuchFileException** – if the specified file does not exist

**writeConfigToStore** ()

Re-writes the config attribute to the job store, so that its values can be retrieved by a separate JobStore instance. No value is returned from this method.

**writeFile** (*localFilePath, jobStoreID=None*)

Takes a file (as a path) and places it in this job store. Returns an ID that can be used to retrieve the file at a later time.

**Parameters**

- **localFilePath** (*str*) – the path to the local file that will be uploaded to the job store.

- **jobStoreID** (*str/None*) – If specified the file will be associated with that job and when `jobStore.delete(job)` is called all files written with the given `job.jobStoreID` will be removed from the job store.

**Raises**

- **ConcurrentFileModificationException** – if the file was modified concurrently during an invocation of this method
- **NoSuchJobException** – if the job specified via `jobStoreID` does not exist

FIXME: some implementations may not raise this

**Returns** an ID referencing the newly created file and can be used to read the file in the future.

**Return type** `str`

**writeFileStream** (*\*args, \*\*kws*)

Similar to `writeFile`, but returns a context manager yielding a tuple of 1) a file handle which can be written to and 2) the ID of the resulting file in the job store. The yielded file handle does not need to and should not be closed explicitly.

**Parameters** **jobStoreID** (*str*) – the id of a job, or `None`. If specified, the file will be associated with that job and when `jobStore.delete(job)` is called all files written with the given `job.jobStoreID` will be removed from the job store.

**Raises**

- **ConcurrentFileModificationException** – if the file was modified concurrently during an invocation of this method
- **NoSuchJobException** – if the job specified via `jobStoreID` does not exist

FIXME: some implementations may not raise this

**Returns** an ID that references the newly created file and can be used to read the file in the future.

**Return type** `str`

**writeSharedFileStream** (*\*args, \*\*kws*)

Returns a context manager yielding a writable file handle to the global file referenced by the given name.

**Parameters**

- **sharedFileName** (*str*) – A file name matching `AbstractJobStore.fileNameRegex`, unique within this job store
- **isProtected** (*bool*) – True if the file must be encrypted, `None` if it may be encrypted or False if it must be stored in the clear.

**Raises** **ConcurrentFileModificationException** – if the file was modified concurrently during an invocation of this method

**writeStatsAndLogging** (*statsAndLoggingString*)

Adds the given statistics/logging string to the store of statistics info.

**Parameters** **statsAndLoggingString** (*str*) – the string to be written to the stats file

**Raises** **ConcurrentFileModificationException** – if the file was modified concurrently during an invocation of this method



## 11.1 toil package

### 11.1.1 Subpackages

**toil.batchSystems package**

**Subpackages**

**toil.batchSystems.mesos package**

**Subpackages**

**toil.batchSystems.mesos.test package**

**Module contents**

**Submodules**

**toil.batchSystems.mesos.batchSystem module**

**toil.batchSystems.mesos.conftest module**

**toil.batchSystems.mesos.executor module**

**Module contents**

## Submodules

`toil.batchSystems.abstractBatchSystem` module

`toil.batchSystems.gridengine` module

`toil.batchSystems.jobDispatcher` module

`toil.batchSystems.lsf` module

`toil.batchSystems.lsf.bsub` (*bsubline*)

`toil.batchSystems.lsf.getJobExitCode` (*lsfJobID*)

`toil.batchSystems.lsf.prepareBsub` (*cpu, mem*)

`toil.batchSystems.parasol` module

`toil.batchSystems.parasolTestSupport` module

`toil.batchSystems.singleMachine` module

`toil.batchSystems.slurm` module

## Module contents

`toil.cwl` package

## Submodules

`toil.cwl.conftest` module

`toil.cwl.cwltoil` module

## Module contents

`toil.jobStores` package

## Subpackages

`toil.jobStores.aws` package

## Submodules

`toil.jobStores.aws.jobStore` module

`toil.jobStores.aws.utils` module

## Module contents

### Submodules

#### `toil.jobStores.abstractJobStore` module

`toil.jobStores.abstractJobStore.findJobStoreForUrl(url, export=False)`

Returns the `AbstractJobStore` subclass that supports the given URL.

**Parameters** `url` (`urlparse.ParseResult`) – The given URL

**Return type** `toil.jobStore.AbstractJobStore`

`toil.jobStores.abstractJobStore.getJobStoreClasses(*args)`

Compiles as list of the classes of all jobStores whose dependencies are installed.

Note that job store names must be manually added.

#### `toil.jobStores.azureJobStore` module

#### `toil.jobStores.conftest` module

#### `toil.jobStores.fileJobStore` module

#### `toil.jobStores.googleJobStore` module

#### `toil.jobStores.utils` module

`toil.jobStores.utils.never(exception)`

`toil.jobStores.utils.retry(delays=(0, 1, 1, 4, 16, 64), timeout=300, predicate=<function never>)`

Retry an operation while the failure matches a given predicate and until a given timeout expires, waiting a given amount of time in between attempts. This function is a generator that yields contextmanagers. See doctests below for example usage.

**Parameters**

- **delays** (`Iterable[float]`) – an iterable yielding the time in seconds to wait before each retried attempt, the last element of the iterable will be repeated.
- **timeout** (`float`) – a overall timeout that should not be exceeded for all attempts together. This is a best-effort mechanism only and it won't abort an ongoing attempt, even if the timeout expires during that attempt.
- **predicate** (`Callable[[Exception], bool]`) – a unary callable returning True if another attempt should be made to recover from the given exception. The default value for this parameter will prevent any retries!

**Returns** a generator yielding context managers, one per attempt

**Return type** `Iterator`

Retry for a limited amount of time:

```
>>> true = lambda _: True
>>> false = lambda _: False
>>> i = 0
```

```
>>> for attempt in retry( delays=[0], timeout=.1, predicate=true ):
...     with attempt:
...         i += 1
...         raise RuntimeError('foo')
Traceback (most recent call last):
...
RuntimeError: foo
>>> i > 1
True
```

If timeout is 0, do exactly one attempt:

```
>>> i = 0
>>> for attempt in retry( timeout=0 ):
...     with attempt:
...         i += 1
...         raise RuntimeError( 'foo' )
Traceback (most recent call last):
...
RuntimeError: foo
>>> i
1
```

Don't retry on success:

```
>>> i = 0
>>> for attempt in retry( delays=[0], timeout=.1, predicate=true ):
...     with attempt:
...         i += 1
>>> i
1
```

Don't retry on unless predicate returns True:

```
>>> i = 0
>>> for attempt in retry( delays=[0], timeout=.1, predicate=false):
...     with attempt:
...         i += 1
...         raise RuntimeError( 'foo' )
Traceback (most recent call last):
...
RuntimeError: foo
>>> i
1
```

`toil.jobStores.utils.retry_http` (*delays=(0, 1, 1, 4, 16, 64), timeout=300, predicate=<function retryable\_http\_error>*)

```
>>> i = 0
>>> for attempt in retry_http(timeout=5):
...     with attempt:
...         i += 1
...         raise urllib2.HTTPError('http://www.test.com', '408', 'some message', {}, None)
Traceback (most recent call last):
...
HTTPError: HTTP Error 408: some message
>>> i > 1
True
```

`toil.jobStores.utils.retryable_http_error` (*e*)

## Module contents

### toil.lib package

#### Subpackages

#### toil.lib.encryption package

#### Submodules

#### toil.lib.encryption.conftest module

## Module contents

### Submodules

#### toil.lib.bioio module

`toil.lib.bioio.absSymPath (path)`  
like `os.path.abspath` except it doesn't dereference symlinks

`toil.lib.bioio.addLoggingFileHandler (fileName, rotatingLogging=False)`

`toil.lib.bioio.addLoggingOptions (parser)`

`toil.lib.bioio.getBasicOptionParser (parser=None)`

`toil.lib.bioio.getLogLevelString ()`

`toil.lib.bioio.getRandomAlphaNumericString (length=10)`  
Returns a random alpha numeric string of the given length.

`toil.lib.bioio.getTempFile (suffix='', rootDir=None)`  
Returns a string representing a temporary file, that must be manually deleted

`toil.lib.bioio.getTotalCpuTime ()`  
Gives the total cpu time, including the children.

`toil.lib.bioio.getTotalCpuTimeAndMemoryUsage ()`  
Gives the total cpu time and memory usage of itself and its children.

`toil.lib.bioio.getTotalMemoryUsage ()`  
Gets the amount of memory used by the process and its children.

`toil.lib.bioio.logFile (fileName, printFunction=<bound method Logger.info of <logging.Logger object at 0x7f1d482445d0>>)`  
Writes out a formatted version of the given log file

`toil.lib.bioio.logStream (fileHandle, shortName, printFunction=<bound method Logger.info of <logging.Logger object at 0x7f1d482445d0>>)`  
Writes out a formatted version of the given log stream.

`toil.lib.bioio.makePublicDir (dirName)`  
Makes a given subdirectory if it doesn't already exist, making sure it is public.

`toil.lib.bioio.parseBasicOptions (parser)`  
Setups the standard things from things added by `getBasicOptionParser`.

`toil.lib.bioio.setLogLevel` (*level*, *logger*=<*logging.RootLogger object*>)

Sets the log level to a given string level (like “INFO”). Operates on the root logger by default, but another logger can be specified instead.

`toil.lib.bioio.setLoggingFromOptions` (*options*)

Sets the logging from a dictionary of name/value options.

`toil.lib.bioio.system` (*command*)

A convenience wrapper around `subprocess.check_call` that logs the command before passing it on. The command can be either a string or a sequence of strings. If it is a string `shell=True` will be passed to `subprocess.check_call`.

## toil.lib.spark module

`toil.lib.spark.spawn_spark_cluster` (*job*, *sudo*, *numWorkers*, *cores*=None, *memory*=None, *disk*=None, *overrideLeaderIP*=None)

### Parameters

- **sudo** (*boolean*) – Whether this code should run docker containers with sudo.
- **numWorkers** (*int*) – The number of worker nodes to have in the cluster. Must be greater than or equal to 1.
- **cores** (*int*) – Optional parameter to set the number of cores per node. If not provided, we use the number of cores on the node that launches the service.
- **memory** (*int or string convertible by `bd2k.util.humanize.human2bytes` to an `int`*) – Optional parameter to set the memory requested per node.
- **disk** (*int or string convertible by `bd2k.util.humanize.human2bytes` to an `int`*) – Optional parameter to set the disk requested per node.

## Module contents

### toil.provisioners package

#### Subpackages

#### toil.provisioners.aws package

#### Submodules

#### toil.provisioners.aws.provisioner module

#### Module contents

## Submodules

`toil.provisioners.abstractProvisioner` module

`toil.provisioners.clusterScaler` module

## Module contents

`toil.test` package

## Subpackages

`toil.test.batchSystems` package

## Submodules

`toil.test.batchSystems.batchSystemTest` module

`toil.test.batchSystems.batchSystemTest.childJob` (*job*, *cmd*)

`toil.test.batchSystems.batchSystemTest.count` (*delta*, *file\_path*)

Increments counter file and returns the max number of times the file has been modified. Counter data must be in the form:

concurrent tasks, max concurrent tasks (counter should be initialized to 0,0)

### Parameters

- **delta** (*int*) – increment value
- **file\_path** (*str*) – path to shared counter file

### Return int max concurrent tasks

`toil.test.batchSystems.batchSystemTest.getCounters` (*path*)

`toil.test.batchSystems.batchSystemTest.grandChildJob` (*job*, *cmd*)

`toil.test.batchSystems.batchSystemTest.greatGrandChild` (*cmd*)

`toil.test.batchSystems.batchSystemTest.measureConcurrency` (*filepath*, *sleep\_time*=5)

Run in parallel to determine the number of concurrent tasks. This code was copied from `toil.batchSystemTestMaxCoresSingleMachineBatchSystemTest` :param str filepath: path to counter file :param int sleep\_time: number of seconds to sleep before counting down :return int max concurrency value:

`toil.test.batchSystems.batchSystemTest.parentJob` (*job*, *cmd*)

`toil.test.batchSystems.batchSystemTest.resetCounters` (*path*)

`toil.test.batchSystems.batchSystemTest.tempFileContaining` (*\*args*, *\*\*kwds*)

## Module contents

`toil.test.cwl` package

## Submodules

**toil.test.cwl.confTest module**

**toil.test.cwl.cwlTest module**

**Module contents**

**toil.test.jobStores package**

**Submodules**

**toil.test.jobStores.jobStoreTest module**

**Module contents**

**toil.test.mesos package**

**Submodules**

**toil.test.mesos.helloWorld module** A simple user script for Toil

`toil.test.mesos.helloWorld.hello_world(job)`

`toil.test.mesos.helloWorld.hello_world_child(job, hw)`

`toil.test.mesos.helloWorld.main()`

**toil.test.mesos.mesosTest module**

**toil.test.mesos.stress module**

`toil.test.mesos.stress.main(numJobs)`

`toil.test.mesos.stress.touchFile(fileStore)`

**Module contents**

**toil.test.provisioners package**

**Submodules**

**toil.test.provisioners.clusterScalerTest module**

**Module contents**

**toil.test.sort package**



## Submodules

### **toil.test.sort.lib module**

`toil.test.sort.lib.copySubRangeOfFile` (*inputFile, fileStart, fileEnd, outputFileHandle*)

Copies the range (in bytes) between fileStart and fileEnd to the given output file handle.

`toil.test.sort.lib.getMidPoint` (*file, fileStart, fileEnd*)

Finds the point in the file to split. Returns an int *i* such that `fileStart <= i < fileEnd`

`toil.test.sort.lib.merge` (*fileHandle1, fileHandle2, outputFileHandle*)

Merges together two files maintaining sorted order.

`toil.test.sort.lib.sort` (*file*)

Sorts the given file.

**toil.test.sort.sort module** A demonstration of toil. Sorts the lines of a file into ascending order by doing a parallel merge sort.

`toil.test.sort.sort.cleanup` (*job, tempOutputFileStoreID, outputFile, cores=1, memory=1048576000, disk='3G'*)

Copies back the temporary file to input once we've successfully sorted the temporary file.

`toil.test.sort.sort.down` (*job, inputFileStoreID, N, downCheckpoints, memory=1048576000*)

Input is a file and a range into that file to sort and an output location in which to write the sorted file. If the range is larger than a threshold *N* the range is divided recursively and a follow on job is then created which merges back the results else the file is sorted and placed in the output.

`toil.test.sort.sort.main` ()

`toil.test.sort.sort.setup` (*job, inputFile, N, downCheckpoints*)

Sets up the sort.

`toil.test.sort.sort.up` (*job, inputFileID1, inputFileID2, memory=1048576000*)

Merges the two files and places them in the output.

### **toil.test.sort.sortTest module**

`toil.test.sort.sortTest.makeFileToSort` (*fileName, lines=10, lineLen=10*)

## Module contents

### **toil.test.src package**

## Submodules

### **toil.test.src.helloWorldTest module**

`toil.test.src.helloWorldTest.childFn` (*job*)

### **toil.test.src.importExportFileTest module**

#### **toil.test.src.jobCacheEjectionTest module**

`toil.test.src.jobCacheEjectionTest.fileTestCache` (*job, file\_a, file\_b, file\_c, logfile*)

Test job exercises Job.FileStore functions

`toil.test.src.jobCacheEjectionTest.fileTestJob` (*job, file\_size*)

Test job exercises Job.FileStore functions

#### **toil.test.src.jobCacheTest module**

#### **toil.test.src.jobEncapsulationTest module**

`toil.test.src.jobEncapsulationTest.encapsulatedJobFn` (*job, string, outFile*)

#### **toil.test.src.jobFileStoreTest module**

`toil.test.src.jobFileStoreTest.fileTestJob` (*job, inputFileStoreIDs, testStrings, chainLength*)

Test job exercises Job.FileStore functions

#### **toil.test.src.jobServiceTest module**

`toil.test.src.jobServiceTest.serviceAccessor` (*job, communicationFiles, outFile, randInt*)

Writes a random integer *i* into the `inJobStoreFileID` file, then tries 10 times reading from `outJobStoreFileID` to get a pair of integers, the first equal to *i* the second written into the output file.

`toil.test.src.jobServiceTest.serviceTest` (*job, outFile, messageInt*)

Creates one service and one accessing job, which communicate with two files to establish that both run concurrently.

`toil.test.src.jobServiceTest.serviceTestParallelRecursive` (*job, outFiles, messageBundles*)

Creates multiple chains of services and accessing jobs.

`toil.test.src.jobServiceTest.serviceTestRecursive` (*job, outFile, messages*)

Creates a chain of services and accessing jobs, each paired together.

#### **toil.test.src.jobTest module**

`toil.test.src.jobTest.fn1Test` (*string, outputFile, promises=[]*)

Function appends string to output file, then returns the next ascii character of the first character in the string, e.g. if string is "AA" returns "B"

`toil.test.src.jobTest.fn2Test` (*pStrings, s, outputFile*)

Function concatenates the strings in *pStrings* and *s*, in that order, and writes the result to the output file. Returns *s*.

#### **toil.test.src.jobWrapperTest module**

#### **toil.test.src.multipartTransferTest module**

#### **toil.test.src.promisedRequirementTest module**

`toil.test.src.promisedRequirementTest.getOne` ()

`toil.test.src.promisedRequirementTest.getThirtyTwoMb` ()

`toil.test.src.promisedRequirementTest.maxConcurrency` (*job, cpuCount, filename, coresPerJob*)

Returns the max number of concurrent tasks when using a `PromisedRequirement` instance to allocate the number of cores per job.

### Parameters

- **cpuCount** (*int*) – number of available cpus
- **filename** (*str*) – path to counter file
- **coresPerJob** (*int*) – number of cores assigned to each job

**Return** int max concurrency value

### **toil.test.src.promisesTest module**

`toil.test.src.promisesTest.child()`  
`toil.test.src.promisesTest.parent(job)`

### **toil.test.src.realtimeLoggerTest module**

### **toil.test.src.regularLogTest module**

### **toil.test.src.resourceTest module**

### **toil.test.src.retainTempDirTest module**

`toil.test.src.retainTempDirTest.tempFileTestErrorJob(job)`  
`toil.test.src.retainTempDirTest.tempFileTestJob(job)`

### **toil.test.src.systemTest module**

### **toil.test.src.toilContextManagerTest module**

`toil.test.src.toilContextManagerTest.childFn(job)`

### **toil.test.src.userDefinedJobArgTypeTest module**

`toil.test.src.userDefinedJobArgTypeTest.jobFunction(job, level, foo)`  
`toil.test.src.userDefinedJobArgTypeTest.main()`

## **Module contents**

### **toil.test.utils package**

## **Submodules**

### **toil.test.utils.utilsTest module**

`toil.test.utils.utilsTest.printUnicodeCharacter()`

## **Module contents**

## Module contents

`toil.test.experimental` (*test\_item*)

Use this to decorate experimental or brittle tests in order to skip them during regular builds.

`toil.test.file_begins_with` (*path*, *prefix*)

`toil.test.make_tests` (*generalMethod*, *targetClass=None*, *\*\*kwargs*)

This method dynamically generates test methods using the `generalMethod` as a template. Each generated function is the result of a unique combination of parameters applied to the `generalMethod`. Each of the parameters has a corresponding string that will be used to name the method. These generated functions are named in the scheme:

`test_[generalMethodName]__[firstParamaterName]_[someValueName]__[secondParamaterName]...`

The arguments following the `generalMethodName` should be a series of one or more dictionaries of the form `{str: type, ...}` where the key represents the name of the value. The names will be used to represent the permutation of values passed for each parameter in the `generalMethod`.

### Parameters

- **generalMethod** – A method that will be parametrized with values passed as kwargs. Note that the `generalMethod` must be a regular method.
- **targetClass** – This represents the class to which the generated test methods will be bound. If no `targetClass` is specified the class of the `generalMethod` is assumed the target.
- **kwargs** – a series of dictionaries defining values, and their respective names where each keyword is the name of a parameter in `generalMethod`.

```
>>> class Foo:
...     def has(self, num, letter):
...         return num, letter
...
...     def hasOne(self, num):
...         return num
```

```
>>> class Bar(Foo):
...     pass
```

```
>>> make_tests(Foo.has, targetClass=Bar, num={'one':1, 'two':2}, letter={'a':'a', 'b':'b'})
```

```
>>> b = Bar()
```

```
>>> assert b.test_has__num_one__letter_a() == b.has(1, 'a')
```

```
>>> assert b.test_has__num_one__letter_b() == b.has(1, 'b')
```

```
>>> assert b.test_has__num_two__letter_a() == b.has(2, 'a')
```

```
>>> assert b.test_has__num_two__letter_b() == b.has(2, 'b')
```

```
>>> f = Foo()
```

```
>>> hasattr(f, 'test_has__num_one__letter_a') # should be false because Foo has no test methods
False
```

```
>>> make_tests(Foo.has, num={'one':1, 'two':2}, letter={'a':'a', 'b':'b'})
```

```
>>> hasattr(f, 'test_has__num_one__letter_a')
True
```

```
>>> assert f.test_has__num_one__letter_a() == f.has(1, 'a')
```

```
>>> assert f.test_has__num_one__letter_b() == f.has(1, 'b')
```

```
>>> assert f.test_has__num_two__letter_a() == f.has(2, 'a')
```

```
>>> assert f.test_has__num_two__letter_b() == f.has(2, 'b')
```

```
>>> make_tests(Foo.hasOne, num={'one':1, 'two':2})
```

```
>>> assert f.test_hasOne__num_one() == f.hasOne(1)
```

```
>>> assert f.test_hasOne__num_two() == f.hasOne(2)
```

`toil.test.needs_aws(test_item)`

Use as a decorator before test classes or methods to only run them if AWS usable.

`toil.test.needs_azure(test_item)`

Use as a decorator before test classes or methods to only run them if Azure is usable.

`toil.test.needs_cwl(test_item)`

Use as a decorator before test classes or methods to only run them if CWLTool is installed and configured.

`toil.test.needs_encryption(test_item)`

Use as a decorator before test classes or methods to only run them if PyNaCl is installed and configured.

`toil.test.needs_google(test_item)`

Use as a decorator before test classes or methods to only run them if Google Storage usable.

`toil.test.needs_gridengine(test_item)`

Use as a decorator before test classes or methods to only run them if GridEngine is installed.

`toil.test.needs_mesos(test_item)`

Use as a decorator before test classes or methods to only run them if the Mesos is installed and configured.

`toil.test.needs_parasol(test_item)`

Use as decorator so tests are only run if Parasol is installed.

`toil.test.needs_slurm(test_item)`

Use as a decorator before test classes or methods to only run them if Slurm is installed.

`toil.test.needs_spark(test_item)`

Use as a decorator before test classes or methods to only run them if Spark is usable.

## toil.utils package

### Submodules

#### toil.utils.toilClean module

Removes the JobStore from a toil run.

`toil.utils.toilClean.main()`

Removes the JobStore from a toil run.

**toil.utils.toilKill module**

Kills any running jobs trees in a rogue toil.

```
toil.utils.toilKill.main()
```

**toil.utils.toilMain module**

```
toil.utils.toilMain.loadModules()
```

```
toil.utils.toilMain.main()
```

```
toil.utils.toilMain.printHelp(modules)
```

**toil.utils.toilStats module**

Reports data about the given toil run.

```
toil.utils.toilStats.buildElement(element, items, itemName)
    Create an element for output.
```

```
toil.utils.toilStats.checkOptions(options, parser)
    Check options, throw parser.error() if something goes wrong
```

```
toil.utils.toilStats.computeColumnWidths(job_types, worker, job, options)
    Return a ColumnWidths() object with the correct max widths.
```

```
toil.utils.toilStats.createSummary(element, containingItems, containingItemName, getFn)
```

```
toil.utils.toilStats.decorateSubHeader(title, columnWidths, options)
    Add a marker to the correct field if the TITLE is sorted on.
```

```
toil.utils.toilStats.decorateTitle(title, options)
    Add a marker to TITLE if the TITLE is sorted on.
```

```
toil.utils.toilStats.get(tree, name)
    Return a float value attribute NAME from TREE.
```

```
toil.utils.toilStats.getStats(options)
    Collect and return the stats and config data.
```

```
toil.utils.toilStats.initializeOptions(parser)
```

```
toil.utils.toilStats.main()
    Reports stats on the workflow, use with -stats option to toil.
```

```
toil.utils.toilStats.padStr(s, field=None)
    Pad the beginning of a string with spaces, if necessary.
```

```
toil.utils.toilStats.prettyMemory(k, field=None, isBytes=False)
    Given input k as kilobytes, return a nicely formatted string.
```

```
toil.utils.toilStats.prettyTime(t, field=None)
    Given input t as seconds, return a nicely formatted string.
```

```
toil.utils.toilStats.printJson(elem)
    Return a JSON formatted string
```

```
toil.utils.toilStats.processData(config, stats, options)
```

```

toil.utils.toilStats.refineData (root, options)
    walk down from the root and gather up the important bits.

toil.utils.toilStats.reportData (tree, options)

toil.utils.toilStats.reportMemory (k, options, field=None, isBytes=False)
    Given k kilobytes, report back the correct format as string.

toil.utils.toilStats.reportNumber (n, options, field=None)
    Given n an integer, report back the correct format as string.

toil.utils.toilStats.reportPrettyData (root, worker, job, job_types, options)
    print the important bits out.

toil.utils.toilStats.reportTime (t, options, field=None)
    Given t seconds, report back the correct format as string.

toil.utils.toilStats.sortJobs (jobTypes, options)
    Return a jobTypes all sorted.

toil.utils.toilStats.sprintTag (key, tag, options, columnWidths=None)
    Generate a pretty-print ready string from a JTag().

toil.utils.toilStats.updateColumnWidths (tag, cw, options)
    Update the column width attributes for this tag's fields.

```

#### toil.utils.toilStatus module

Reports the state of a Toil workflow

```

toil.utils.toilStatus.main()
    Reports the state of the toil.

```

#### Module contents

### 11.1.2 Submodules

#### 11.1.3 toil.common module

```

toil.common.addOptions (parser, config=<toil.common.Config object>)
    Adds toil options to a parser object, either optparse or argparse.

```

```

toil.common.cacheDirName (workflowID)

    Returns Name of the cache directory.

```

```

toil.common.parseSetEnv (l)
    Parses a list of strings of the form "NAME=VALUE" or just "NAME" into a dictionary. Strings of the latter
    from will result in dictionary entries whose value is None.

```

**Return type** dict[str,str]

```

>>> parseSetEnv([])
{}
>>> parseSetEnv(['a'])
{'a': None}
>>> parseSetEnv(['a='])
{'a': ''}
>>> parseSetEnv(['a=b'])

```

```
{'a': 'b'}
>>> parseSetEnv(['a=a', 'a=b'])
{'a': 'b'}
>>> parseSetEnv(['a=b', 'c=d'])
{'a': 'b', 'c': 'd'}
>>> parseSetEnv(['a=b=c'])
{'a': 'b=c'}
>>> parseSetEnv([''])
Traceback (most recent call last):
...
ValueError: Empty name
>>> parseSetEnv(['=1'])
Traceback (most recent call last):
...
ValueError: Empty name
```

### 11.1.4 toil.job module

### 11.1.5 toil.jobWrapper module

### 11.1.6 toil.leader module

The leader script (of the leader/worker pair) for running jobs.

`toil.leader.innerLoop` (*jobStore*, *config*, *batchSystem*, *toilState*, *jobBatcher*, *serviceManager*, *statsAndLogging*)

#### Parameters

- **jobStore** (`toil.jobStores.abstractJobStore.AbstractJobStore`) –
- **config** (`toil.common.Config`) –
- **batchSystem** (`toil.batchSystems.abstractBatchSystem.AbstractBatchSystem`) –
- **toilState** (`ToilState`) –
- **jobBatcher** (`JobBatcher`) –
- **serviceManager** (`ServiceManager`) –
- **statsAndLogging** (`StatsAndLogging`) –

`toil.leader.mainLoop` (*config*, *batchSystem*, *provisioner*, *jobStore*, *rootJobWrapper*, *jobCache=None*)

This is the main loop from which jobs are issued and processed.

If *jobCache* is passed, it must be a dict from job ID to pre-existing `JobWrapper` objects. Jobs will be loaded from the cache (which can be downloaded from the *jobStore* in a batch).

**Raises** `toil.leader.FailedJobsException` if at the end of function their remain failed jobs

**Returns** The return value of the root job's run function.

**Return type** Any

### 11.1.7 toil.realtimeLogger module

Implements a real-time UDP-based logging system that user scripts can use for debugging.



### 11.1.8 toil.resource module

### 11.1.9 toil.toilState module

### 11.1.10 toil.version module

### 11.1.11 toil.worker module

`toil.worker.main()`

`toil.worker.nextOpenDescriptor()`

Gets the number of the next available file descriptor.

### 11.1.12 Module contents

`toil.inVirtualEnv()`

`toil.physicalMemory(*args)`

```
>>> n = physicalMemory()
>>> n > 0
True
>>> n == physicalMemory()
True
```

`toil.resolveEntryPoint(entryPoint)`

Returns the path to the given entry point (see `setup.py`) that *should* work on a worker. The return value may be an absolute or a relative path.

`toil.toilPackageDirPath()`

Returns the absolute path of the directory that corresponds to the top-level toil package. The return value is guaranteed to end in `'/toil'`.



---

## Indices and tables

---

- `genindex`
- `modindex`
- `search`



## t

- toil, 77
- toil.batchSystems, 62
- toil.batchSystems.abstractBatchSystem, 62
- toil.batchSystems.gridengine, 62
- toil.batchSystems.jobDispatcher, 62
- toil.batchSystems.lsf, 62
- toil.batchSystems.mesos, 61
- toil.batchSystems.mesos.conftest, 61
- toil.batchSystems.mesos.test, 61
- toil.batchSystems.parasol, 62
- toil.batchSystems.parasolTestSupport, 62
- toil.batchSystems.singleMachine, 62
- toil.batchSystems.slurm, 62
- toil.common, 75
- toil.cwl, 62
- toil.cwl.conftest, 62
- toil.job, 76
- toil.jobStores, 65
- toil.jobStores.abstractJobStore, 63
- toil.jobStores.aws, 63
- toil.jobStores.conftest, 63
- toil.jobStores.fileJobStore, 63
- toil.jobStores.utils, 63
- toil.jobWrapper, 76
- toil.leader, 76
- toil.lib, 66
- toil.lib.bioio, 65
- toil.lib.encryption, 65
- toil.lib.encryption.conftest, 65
- toil.lib.spark, 66
- toil.provisioners, 67
- toil.provisioners.abstractProvisioner, 67
- toil.provisioners.aws, 66
- toil.provisioners.clusterScaler, 67
- toil.realtimeLogger, 76
- toil.resource, 77
- toil.test, 72
- toil.test.batchSystems, 67
- toil.test.batchSystems.batchSystemTest, 67
- toil.test.cwl, 68
- toil.test.cwl.conftest, 68
- toil.test.cwl.cwlTest, 68
- toil.test.jobStores, 68
- toil.test.mesos, 68
- toil.test.mesos.helloWorld, 68
- toil.test.mesos.mesosTest, 68
- toil.test.mesos.stress, 68
- toil.test.provisioners, 68
- toil.test.provisioners.clusterScalerTest, 68
- toil.test.sort, 69
- toil.test.sort.lib, 69
- toil.test.sort.sort, 69
- toil.test.sort.sortTest, 69
- toil.test.src, 71
- toil.test.src.helloWorldTest, 69
- toil.test.src.importExportFileTest, 69
- toil.test.src.jobCacheEjectionTest, 70
- toil.test.src.jobCacheTest, 70
- toil.test.src.jobEncapsulationTest, 70
- toil.test.src.jobFileStoreTest, 70
- toil.test.src.jobServiceTest, 70
- toil.test.src.jobTest, 70
- toil.test.src.jobWrapperTest, 70
- toil.test.src.promisedRequirementTest, 70
- toil.test.src.promisesTest, 71
- toil.test.src.realtimeLoggerTest, 71
- toil.test.src.regularLogTest, 71
- toil.test.src.resourceTest, 71
- toil.test.src.retainTempDirTest, 71
- toil.test.src.systemTest, 71
- toil.test.src.toilContextManagerTest, 71
- toil.test.src.userDefinedJobArgTypeTest, 71

`toil.test.utils`, [71](#)  
`toil.test.utils.utilsTest`, [71](#)  
`toil.toilState`, [77](#)  
`toil.utils`, [75](#)  
`toil.utils.toilClean`, [73](#)  
`toil.utils.toilKill`, [74](#)  
`toil.utils.toilMain`, [74](#)  
`toil.utils.toilStats`, [74](#)  
`toil.utils.toilStatus`, [75](#)  
`toil.version`, [77](#)  
`toil.worker`, [77](#)

## A

absSymPath() (in module toil.lib.bioio), 65  
 AbstractBatchSystem (class  
   toil.batchSystems.abstractBatchSystem), 53  
 AbstractJobStore (class  
   toil.jobStores.abstractJobStore), 55  
 addChild() (toil.job.EncapsulatedJob method), 46  
 addChild() (toil.job.Job method), 37  
 addChildFn() (toil.job.Job method), 37  
 addChildJobFn() (toil.job.Job method), 38  
 addFollowOn() (toil.job.EncapsulatedJob method), 46  
 addFollowOn() (toil.job.Job method), 38  
 addFollowOnFn() (toil.job.Job method), 38  
 addFollowOnJobFn() (toil.job.Job method), 38  
 addLoggingFileHandler() (in module toil.lib.bioio), 65  
 addLoggingOptions() (in module toil.lib.bioio), 65  
 addOptions() (in module toil.common), 75  
 addService() (toil.job.EncapsulatedJob method), 46  
 addService() (toil.job.Job method), 38  
 addToilOptions() (toil.job.Job.Runner static method), 43  
 allocatePromiseFile() (toil.job.Job method), 39  
 asyncWrite() (toil.job.Job.FileStore method), 41

## B

bsub() (in module toil.batchSystems.lsf), 62  
 buildElement() (in module toil.utils.toilStats), 74

## C

cacheDirName() (in module toil.common), 75  
 check() (toil.job.Job.Service method), 45  
 checkJobGraphAcyclic() (toil.job.Job method), 39  
 checkJobGraphConnected() (toil.job.Job method), 39  
 checkJobGraphForDeadlocks() (toil.job.Job method), 39  
 checkNewCheckpointsAreLeafVertices() (toil.job.Job method), 39  
 checkOptions() (in module toil.utils.toilStats), 74  
 child() (in module toil.test.src.promisesTest), 71  
 childFn() (in module toil.test.src.helloWorldTest), 69

childFn() (in module toil.test.src.toilContextManagerTest), 71  
 in childJob() (in module  
   toil.test.batchSystems.batchSystemTest), 67  
 in clean() (toil.jobStores.abstractJobStore.AbstractJobStore method), 55  
 cleanup() (in module toil.test.sort.sort), 69  
 computeColumnWidths() (in module toil.utils.toilStats), 74  
 config (toil.jobStores.abstractJobStore.AbstractJobStore attribute), 55  
 convertPromises() (toil.job.PromisedRequirement static method), 47  
 copySubRangeOfFile() (in module toil.test.sort.lib), 69  
 count() (in module toil.test.batchSystems.batchSystemTest), 67  
 create() (toil.jobStores.abstractJobStore.AbstractJobStore method), 55  
 createBatchSystem() (toil.common.Toil static method), 44  
 createRootJob() (toil.jobStores.abstractJobStore.AbstractJobStore method), 56  
 createSummary() (in module toil.utils.toilStats), 74

## D

decorateSubHeader() (in module toil.utils.toilStats), 74  
 decorateTitle() (in module toil.utils.toilStats), 74  
 delete() (toil.jobStores.abstractJobStore.AbstractJobStore method), 56  
 deleteFile() (toil.jobStores.abstractJobStore.AbstractJobStore method), 56  
 deleteGlobalFile() (toil.job.Job.FileStore method), 41  
 deleteJobStore() (toil.jobStores.abstractJobStore.AbstractJobStore method), 56  
 down() (in module toil.test.sort.sort), 69

## E

effectiveRequirements() (toil.job.Job method), 39  
 encapsulate() (toil.job.Job method), 39

EncapsulatedJob (class in toil.job), 46  
 encapsulatedJobFn() (in module  
 toil.test.src.jobEncapsulationTest), 70  
 exists() (toil.jobStores.abstractJobStore.AbstractJobStore  
 method), 56  
 experimental() (in module toil.test), 72  
 exportFile() (toil.common.Toil method), 44  
 exportFile() (toil.job.Job.FileStore method), 41  
 exportFile() (toil.jobStores.abstractJobStore.AbstractJobStore  
 method), 56

## F

file\_begins\_with() (in module toil.test), 72  
 fileExists() (toil.jobStores.abstractJobStore.AbstractJobStore  
 method), 56  
 filesToDelete (toil.job.Promise attribute), 47  
 fileTestCache() (in module  
 toil.test.src.jobCacheEjectionTest), 70  
 fileTestJob() (in module  
 toil.test.src.jobCacheEjectionTest), 70  
 fileTestJob() (in module toil.test.src.jobFileStoreTest), 70  
 findJobStoreForUrl() (in module  
 toil.jobStores.abstractJobStore), 63  
 fn1Test() (in module toil.test.src.jobTest), 70  
 fn2Test() (in module toil.test.src.jobTest), 70  
 FunctionWrappingJob (class in toil.job), 45

## G

get() (in module toil.utils.toilStats), 74  
 getBasicOptionParser() (in module toil.lib.bioio), 65  
 getCounters() (in module  
 toil.test.batchSystems.batchSystemTest),  
 67  
 getDefaultArgumentParser() (toil.job.Job.Runner static  
 method), 43  
 getDefaultOptions() (toil.job.Job.Runner static method),  
 43  
 getEmptyFileStoreID() (toil.jobStores.abstractJobStore.AbstractJobStore  
 method), 56  
 getEnv() (toil.jobStores.abstractJobStore.AbstractJobStore  
 method), 56  
 getIssuedBatchJobIDs() (toil.batchSystems.abstractBatchSystem.AbstractBatchSystem  
 method), 53  
 getjobexitcode() (in module toil.batchSystems.lsf), 62  
 getJobStoreClasses() (in module  
 toil.jobStores.abstractJobStore), 63  
 getLocalTempDir() (toil.job.Job.FileStore method), 41  
 getLocalTempFile() (toil.job.Job.FileStore method), 41  
 getLocalTempFileName() (toil.job.Job.FileStore  
 method), 42  
 getLogLevelString() (in module toil.lib.bioio), 65  
 getMidPoint() (in module toil.test.sort.lib), 69  
 getOne() (in module toil.test.src.promisedRequirementTest),  
 70

getPublicUrl() (toil.jobStores.abstractJobStore.AbstractJobStore  
 method), 57  
 getRandomAlphaNumericString() (in module  
 toil.lib.bioio), 65  
 getRescueBatchJobFrequency()  
 (toil.batchSystems.abstractBatchSystem.AbstractBatchSystem  
 class method), 53  
 getRootJobs() (toil.job.Job method), 40  
 getRunningBatchJobIDs()  
 (toil.batchSystems.abstractBatchSystem.AbstractBatchSystem  
 method), 53  
 getSharedPublicUrl() (toil.jobStores.abstractJobStore.AbstractJobStore  
 method), 57  
 getStats() (in module toil.utils.toilStats), 74  
 getTempFile() (in module toil.lib.bioio), 65  
 getThirtyTwoMb() (in module  
 toil.test.src.promisedRequirementTest), 70  
 getTopologicalOrderingOfJobs() (toil.job.Job method),  
 40  
 getTotalCpuTime() (in module toil.lib.bioio), 65  
 getTotalCpuTimeAndMemoryUsage() (in module  
 toil.lib.bioio), 65  
 getTotalMemoryUsage() (in module toil.lib.bioio), 65  
 getUpdatedBatchJob() (toil.batchSystems.abstractBatchSystem.AbstractBatchSystem  
 method), 53  
 getUserScript() (toil.job.FunctionWrappingJob method),  
 45  
 getUserScript() (toil.job.Job method), 40  
 getValue() (toil.job.PromisedRequirement method), 47  
 getWorkflowDir() (toil.common.Toil static method), 44  
 grandChildJob() (in module  
 toil.test.batchSystems.batchSystemTest),  
 67  
 greatGrandChild() (in module  
 toil.test.batchSystems.batchSystemTest),  
 67

## H

hasChild() (toil.job.Job method), 40  
 hello\_world() (in module toil.test.mesos.helloWorld), 68  
 hello\_world\_child() (in module  
 toil.test.mesos.helloWorld), 68

## I

importFile() (toil.common.Toil method), 44  
 importFile() (toil.job.Job.FileStore method), 42  
 importFile() (toil.jobStores.abstractJobStore.AbstractJobStore  
 method), 57  
 initializeOptions() (in module toil.utils.toilStats), 74  
 innerLoop() (in module toil.leader), 76  
 inVirtualEnv() (in module toil), 77  
 issueBatchJob() (toil.batchSystems.abstractBatchSystem.AbstractBatchSystem  
 method), 53



**J**

Job (class in toil.job), 37  
 Job.FileStore (class in toil.job), 41  
 Job.Runner (class in toil.job), 43  
 Job.Service (class in toil.job), 45  
 JobException (class in toil.job), 47  
 jobFunction() (in module toil.test.src.userDefinedJobArgTypeTest), 71  
 JobFunctionWrappingJob (class in toil.job), 46  
 JobGraphDeadlockException (class in toil.job), 47  
 jobs() (toil.jobStores.abstractJobStore.AbstractJobStore method), 57

**K**

killBatchJobs() (toil.batchSystems.abstractBatchSystem.AbstractBatchSystem method), 54

**L**

load() (toil.jobStores.abstractJobStore.AbstractJobStore method), 58  
 loadModules() (in module toil.utils.toilMain), 74  
 loadOrCreateJobStore() (toil.common.Toil static method), 44  
 loadRootJob() (toil.jobStores.abstractJobStore.AbstractJobStore method), 58  
 logFile() (in module toil.lib.bioio), 65  
 logStream() (in module toil.lib.bioio), 65  
 logToMaster() (toil.job.Job.FileStore method), 42

**M**

main() (in module toil.test.mesos.helloWorld), 68  
 main() (in module toil.test.mesos.stress), 68  
 main() (in module toil.test.sort.sort), 69  
 main() (in module toil.test.src.userDefinedJobArgTypeTest), 71  
 main() (in module toil.utils.toilClean), 73  
 main() (in module toil.utils.toilKill), 74  
 main() (in module toil.utils.toilMain), 74  
 main() (in module toil.utils.toilStats), 74  
 main() (in module toil.utils.toilStatus), 75  
 main() (in module toil.worker), 77  
 mainLoop() (in module toil.leader), 76  
 make\_tests() (in module toil.test), 72  
 makeFileToSort() (in module toil.test.sort.sortTest), 69  
 makePublicDir() (in module toil.lib.bioio), 65  
 maxConcurrency() (in module toil.test.src.promisedRequirementTest), 70  
 measureConcurrency() (in module toil.test.batchSystems.batchSystemTest), 67  
 merge() (in module toil.test.sort.lib), 69

**N**

needs\_aws() (in module toil.test), 73  
 needs\_azure() (in module toil.test), 73  
 needs\_cwl() (in module toil.test), 73  
 needs\_encryption() (in module toil.test), 73  
 needs\_google() (in module toil.test), 73  
 needs\_gridengine() (in module toil.test), 73  
 needs\_mesos() (in module toil.test), 73  
 needs\_parasol() (in module toil.test), 73  
 needs\_slurm() (in module toil.test), 73  
 needs\_spark() (in module toil.test), 73  
 never() (in module toil.jobStores.utils), 63  
 nextOpenDescriptor() (in module toil.worker), 77

**O**

openBatchJob() (toil.jobStores.abstractJobStore.AbstractJobStore method), 42

**P**

padStr() (in module toil.utils.toilStats), 74  
 parent() (in module toil.test.src.promisesTest), 71  
 parentJob() (in module toil.test.batchSystems.batchSystemTest), 67  
 parseBasicOptions() (in module toil.lib.bioio), 65  
 parseSetEnv() (in module toil.common), 75  
 physicalMemory() (in module toil), 77  
 prepareBsub() (in module toil.batchSystems.lsf), 62  
 prettyMemory() (in module toil.utils.toilStats), 74  
 prettyTime() (in module toil.utils.toilStats), 74  
 printHelp() (in module toil.utils.toilMain), 74  
 printJson() (in module toil.utils.toilStats), 74  
 printUnicodeCharacter() (in module toil.test.utils.utilsTest), 71  
 processData() (in module toil.utils.toilStats), 74  
 Promise (class in toil.job), 46  
 PromisedRequirement (class in toil.job), 47  
 publicUrlExpiration (toil.jobStores.abstractJobStore.AbstractJobStore attribute), 58

**R**

readFile() (toil.jobStores.abstractJobStore.AbstractJobStore method), 58  
 readFileStream() (toil.jobStores.abstractJobStore.AbstractJobStore method), 58  
 readGlobalFile() (toil.job.Job.FileStore method), 42  
 readGlobalFileStream() (toil.job.Job.FileStore method), 42  
 readSharedFileStream() (toil.jobStores.abstractJobStore.AbstractJobStore method), 58  
 readStatsAndLogging() (toil.jobStores.abstractJobStore.AbstractJobStore method), 58  
 refineData() (in module toil.utils.toilStats), 74  
 reportData() (in module toil.utils.toilStats), 75

reportMemory() (in module toil.utils.toilStats), 75  
reportNumber() (in module toil.utils.toilStats), 75  
reportPrettyData() (in module toil.utils.toilStats), 75  
reportTime() (in module toil.utils.toilStats), 75  
resetCounters() (in module  
toil.test.batchSystems.batchSystemTest),  
67  
resolveEntryPoint() (in module toil), 77  
restart() (toil.common.Toil method), 44  
retry() (in module toil.jobStores.utils), 63  
retry\_http() (in module toil.jobStores.utils), 64  
retryable\_http\_error() (in module toil.jobStores.utils), 64  
rootJobStoreIDFileName  
(toil.jobStores.abstractJobStore.AbstractJobStore  
attribute), 59  
run() (toil.job.FunctionWrappingJob method), 45  
run() (toil.job.Job method), 40  
run() (toil.job.JobFunctionWrappingJob method), 46  
rv() (toil.job.EncapsulatedJob method), 46  
rv() (toil.job.Job method), 40

## S

serviceAccessor() (in module  
toil.test.src.jobServiceTest), 70  
serviceTest() (in module toil.test.src.jobServiceTest), 70  
serviceTestParallelRecursive() (in module  
toil.test.src.jobServiceTest), 70  
serviceTestRecursive() (in module  
toil.test.src.jobServiceTest), 70  
setEnv() (toil.batchSystems.abstractBatchSystem.AbstractBatchSystem  
method), 54  
setLoggingFromOptions() (in module toil.lib.bioio), 66  
setLogLevel() (in module toil.lib.bioio), 65  
setRootJob() (toil.jobStores.abstractJobStore.AbstractJobStore  
method), 59  
setup() (in module toil.test.sort.sort), 69  
sharedFileNameRegex (toil.jobStores.abstractJobStore.AbstractJobStore  
attribute), 59  
shutdown() (toil.batchSystems.abstractBatchSystem.AbstractBatchSystem  
method), 54  
sort() (in module toil.test.sort.lib), 69  
sortJobs() (in module toil.utils.toilStats), 75  
spawn\_spark\_cluster() (in module toil.lib.spark), 66  
sprintTag() (in module toil.utils.toilStats), 75  
start() (toil.common.Toil method), 44  
start() (toil.job.Job.Service method), 45  
startToil() (toil.job.Job.Runner static method), 43  
stop() (toil.job.Job.Service method), 45  
supportsHotDeployment()  
(toil.batchSystems.abstractBatchSystem.AbstractBatchSystem  
class method), 54  
supportsWorkerCleanup()  
(toil.batchSystems.abstractBatchSystem.AbstractBatchSystem  
class method), 54

system() (in module toil.lib.bioio), 66

## T

tempFileContaining() (in module  
toil.test.batchSystems.batchSystemTest),  
67  
tempFileTestErrorJob() (in module  
toil.test.src.retainTempDirTest), 71  
tempFileTestJob() (in module  
toil.test.src.retainTempDirTest), 71  
Toil (class in toil.common), 43  
toil (module), 77  
toil.batchSystems (module), 62  
toil.batchSystems.abstractBatchSystem (module), 62  
toil.batchSystems.gridengine (module), 62  
toil.batchSystems.jobDispatcher (module), 62  
toil.batchSystems.lsf (module), 62  
toil.batchSystems.mesos (module), 61  
toil.batchSystems.mesos.confTest (module), 61  
toil.batchSystems.mesos.test (module), 61  
toil.batchSystems.paraSol (module), 62  
toil.batchSystems.paraSolTestSupport (module), 62  
toil.batchSystems.singleMachine (module), 62  
toil.batchSystems.slurm (module), 62  
toil.common (module), 75  
toil.cwl (module), 62  
toil.cwl.confTest (module), 62  
toil.job (module), 76  
toil.jobStores (module), 65  
toil.jobStores.abstractJobStore (module), 63  
toil.jobStores.aws (module), 63  
toil.jobStores.confTest (module), 63  
toil.jobStores.fileJobStore (module), 63  
toil.jobStores.utils (module), 63  
toil.jobWrapper (module), 76  
toil.leader (module), 76  
toil.lib.bioio (module), 65  
toil.lib.bioio.encryption (module), 65  
toil.lib.encryption.confTest (module), 65  
toil.lib.spark (module), 66  
toil.provisioners (module), 67  
toil.provisioners.abstractProvisioner (module), 67  
toil.provisioners.aws (module), 66  
toil.provisioners.clusterScaler (module), 67  
toil.realtimeLogger (module), 76  
toil.resource (module), 77  
toil.test (module), 72  
toil.test.batchSystems (module), 67  
toil.test.batchSystems.batchSystemTest (module), 67  
toil.test.cwl (module), 68  
toil.test.cwl.confTest (module), 68  
toil.test.cwl.cwlTest (module), 68  
toil.test.jobStores (module), 68

toil.test.mesos (module), 68  
toil.test.mesos.helloWorld (module), 68  
toil.test.mesos.mesosTest (module), 68  
toil.test.mesos.stress (module), 68  
toil.test.provisioners (module), 68  
toil.test.provisioners.clusterScalerTest (module), 68  
toil.test.sort (module), 69  
toil.test.sort.lib (module), 69  
toil.test.sort.sort (module), 69  
toil.test.sort.sortTest (module), 69  
toil.test.src (module), 71  
toil.test.src.helloWorldTest (module), 69  
toil.test.src.importExportFileTest (module), 69  
toil.test.src.jobCacheEjectionTest (module), 70  
toil.test.src.jobCacheTest (module), 70  
toil.test.src.jobEncapsulationTest (module), 70  
toil.test.src.jobFileStoreTest (module), 70  
toil.test.src.jobServiceTest (module), 70  
toil.test.src.jobTest (module), 70  
toil.test.src.jobWrapperTest (module), 70  
toil.test.src.promisedRequirementTest (module), 70  
toil.test.src.promisesTest (module), 71  
toil.test.src.realtimeLoggerTest (module), 71  
toil.test.src.regularLogTest (module), 71  
toil.test.src.resourceTest (module), 71  
toil.test.src.retainTempDirTest (module), 71  
toil.test.src.systemTest (module), 71  
toil.test.src.toilContextManagerTest (module), 71  
toil.test.src.userDefinedJobArgTypeTest (module), 71  
toil.test.utils (module), 71  
toil.test.utils.utilsTest (module), 71  
toil.toilState (module), 77  
toil.utils (module), 75  
toil.utils.toilClean (module), 73  
toil.utils.toilKill (module), 74  
toil.utils.toilMain (module), 74  
toil.utils.toilStats (module), 74  
toil.utils.toilStatus (module), 75  
toil.version (module), 77  
toil.worker (module), 77  
toilPackageDirPath() (in module toil), 77  
touchFile() (in module toil.test.mesos.stress), 68

## U

up() (in module toil.test.sort.sort), 69  
update() (toil.jobStores.abstractJobStore.AbstractJobStore method), 59  
updateColumnWidths() (in module toil.utils.toilStats), 75  
updateFile() (toil.jobStores.abstractJobStore.AbstractJobStore method), 59  
updateFileStream() (toil.jobStores.abstractJobStore.AbstractJobStore method), 59

## W

wrapFn() (toil.job.Job static method), 40  
wrapJobFn() (toil.job.Job static method), 40  
writeConfigToStore() (toil.jobStores.abstractJobStore.AbstractJobStore method), 59  
writeFile() (toil.jobStores.abstractJobStore.AbstractJobStore method), 59  
writeFileStream() (toil.jobStores.abstractJobStore.AbstractJobStore method), 60  
writeGlobalFile() (toil.job.Job.FileStore method), 42  
writeGlobalFileStream() (toil.job.Job.FileStore method), 43  
writeSharedFileStream() (toil.jobStores.abstractJobStore.AbstractJobStore method), 60  
writeStatsAndLogging() (toil.jobStores.abstractJobStore.AbstractJobStore method), 60