Arvados - Story #3603

[Crunch] Design good Crunch task API, including considerations about "jobs-within-jobs" and "reusable tasks" ideas

08/14/2014 03:40 PM - Abram Connelly

Status:	Closed	**Start date:**	08/27/2014
Priority:	Normal	**Due date:**	
Assigned To:		**% Done:**	100%
Category:	Crunch	**Estimated time:**	0.00 hour
Target version:			

Description

I find there is a lot of boiler plate code that needs to be written in order to get even a basic crunch job running. As of this writing, there is a 'run-command' script which makes this process a lot easier but even the 'run-command' script is becoming unwieldy and doesn't always allow for easily written crunch jobs.

As a concrete example, I have a pipeline that expects a text file in a collection to have on each line a resource locator string and a "human ID" (comma separated). The first few lines of the file look as follows:

```
01ebb0ee6072d5d1274e5f805c520d38+51822,huEC6EEC
01f2b380198d9f2f8592e3eca2731b00+52431,huC434ED
039a116e865a63956dded36894dc7f20+52432,hu0D879F
04ba952fb67485b6c207db50cf9231eb+52433,huF1DC30
0527805fd792af51b89f7a693fb86f1a+52431,hu032C04
...
```

Each of the locator strings represents a collection with many files that the program in the pipeline will process.

The 'run-command' has a 'task.foreach' capability which can, for each input line, create a new task. This almost does what I want but since I have two fields that I want to pass into my program, I have to do a small processing step to parse the input to be passed into the program I want to run.

Using 'run-command', I have written a 'shim' script that takes in parameters on the command line and the executes the program.

Here is the relevant portion of the template with the 'shim' script put in:

```
            ...
            "script": "run-command",
            "script_parameters": {
                "command": [
                    "$(job.srcdir)/crunch_scripts/shim",
                    "$(fj_human)",
                    "$(job.srcdir)/crunch_scripts/bin/tileruler",
                    "$(file $(REFPATH))",
                    "$(job.srcdir)/crunch_scripts/bin/lz4"
                ],

                "fj_human" : "$(file $(HUMAN_COLLECTION_LIST))",
                "task.foreach": "fj_human",

                "HUMAN_COLLECTION_LIST": {
                    "required": true,
                    "dataclass": "File"
                },

                "REFPATH" : {
                    "required": true,
                    "dataclass": "File"
                }
            },
            ...
```

And the shim script for completeness:

```
#!/bin/bash

fj_human=$1
tileruler=$2
refpath=$3
lz4=$4

fj_uuid=` echo $fj_human | cut -f1 -d',' `
huname=` echo $fj_human | cut -f2 -d',' `

$tileruler --crunch --noterm abv -human $huname -fastj-path $TASK_KEEPMOUNT/$fj_uuid -lib-path $re
fpath
```

One could imagine extending the 'run-command' to include more options to help facilitate this type of workflow, but I think the deeper issue is providing a simpler SDK for common environments.

For example, here is what I would imagine a template looking like:

```
            ...
            "script": "myjob",
            "script_parameters": {
                "HUMAN_COLLECTION_LIST": {
                    "required": true,
                    "dataclass": "File"
                },
                "REFPATH" : {
                    "required": true,
                    "dataclass": "File"
                }
            },
            ...
```

And a hypothetical bash 'myjob' script:

```
#!/usr/bin/arvenv

for x in `cat $HUMAN_COLLECTION_LIST`
do
  fj_uuid=`echo $x | cut -f1 -d,`
  huname=`echo $x | cut -f2 -d,`

  arvqueuetask arvenv tileruler --crunch --noterm abv -human $huname -fastj-path $TASK_KEEPMOUNT/$
fj_uuid -lib-path $REFPATH
done
```

Here is a hypothetical "myjob" Python script to do the same:

```
#!/usr/bin/python

import os
import arvados as arv

job = arv.this_job()
input_collection = job["script_parameters"]["HUMAN_COLLECTION_LIST"]
refpath = job["script_parameters"]["REFPATH"]

with open( input_collection ) as f:
  for line in f.readlines():
    line = line.strip()
    fj_uuid, huname = line.split(',')
    arv.queuetask( [ "arvenv", "tileruler", "--crunch", "--noterm", "abv", "-human", huname, "-fas
tj-path", arv.keepmount() + "/" + fj_uuid, "-lib-path", refpath ] )
```

Where 'arvenv' could be an enhanced version of 'run-command' or something else that's smart about setting up the environment.

Both of the hypothetical scripts might seem a bit short but I believe they are much more in line with what people (and me) expect these type of 'adaptor' scripts to look like.

Making these scripts with the Python SDK would require at least two pages of boiler plate code. The 'run-command' script helps reduce boiler plate but, in my opinion, at the cost of versatility and readability. Both of the above scripts only really need some access to variables passed as specified in the template and need an easily accessible helper functions to arvados functionality which, in this case, is the ability to create tasks easily.

All of the above uses the assumptions that 'run-command' makes, such as making the current working directory the 'output' directory and any files created in the current working directory will be automatically put into a collection at job/task end.

Subtasks:

Task # 5031: Review/feedback	**Closed**
Task # 3718: Hash out desired API with science team	**Closed**

Related issues:

Related to Arvados - Feature #4528: [Crunch] Dynamic task allocation based on...	**Closed**	**11/14/2014**	
Related to Arvados - Feature #4561: [SDKs] Refactor run-command so it can be ...	**New**		
Precedes Arvados - Story #3347: [Crunch] Run dev (and real) jobs using a synt...	**Closed**	**08/28/2014**	**08/28/2014**

Associated revisions

Revision 26aa2f1a - 11/12/2014 02:57 PM - Brett Smith

Merge branch '3603-pysdk-file-api-wip'

Refs #3603. Closes #4316.

History

#1 - 08/15/2014 11:42 AM - Tom Clegg

- Target version set to Arvados Future Sprints

#2 - 08/15/2014 11:45 AM - Tom Clegg

- Subject changed from Crunch jobs are hard to write to [Crunch] Good Crunch task API

- Story points set to 3.0

#3 - 08/27/2014 02:06 PM - Ward Vandewege

- Target version changed from Arvados Future Sprints to 2014-09-17 sprint

#4 - 08/27/2014 02:41 PM - Brett Smith

- Assigned To set to Brett Smith

#5 - 09/17/2014 03:08 PM - Brett Smith

- Target version changed from 2014-09-17 sprint to 2014-10-08 sprint

#6 - 09/17/2014 03:40 PM - Tom Clegg

- Subject changed from [Crunch] Good Crunch task API to [Crunch] Design good Crunch task API, including considerations about "jobs-within-jobs" and "reusable tasks" ideas

#7 - 09/23/2014 02:23 PM - Brett Smith

I showed the current draft to Sally. This diff is the outcome. Reaction was generally positive (she said she could write all her current scripts this way). Some of the diff is minor example bugs/readability stuff. One thing that isn't clear enough is the rules for parallelization. That probably needs to be hashed out more.

#8 - 10/08/2014 05:52 PM - Ward Vandewege

- Target version changed from 2014-10-08 sprint to 2014-10-29 sprint

#9 - 10/24/2014 09:38 PM - Brett Smith

3603-pysdk-file-api-wip is up for review. It adds open() methods to CollectionReader and CollectionWriter that act like the built-in, returning file-like objects that should be familiar to Python programmers.

#10 - 10/29/2014 05:48 PM - Ward Vandewege

- Target version changed from 2014-10-29 sprint to 2014-11-19 sprint

#11 - 10/29/2014 06:45 PM - Tom Clegg

- Target version changed from 2014-11-19 sprint to Arvados Future Sprints

#12 - 12/10/2014 08:13 PM - Brett Smith

- Target version changed from Arvados Future Sprints to 2015-01-07 sprint

#13 - 01/05/2015 05:07 PM - Brett Smith

- Target version changed from 2015-01-07 sprint to Arvados Future Sprints

#14 - 01/07/2015 08:24 PM - Tom Clegg

- Target version changed from Arvados Future Sprints to 2015-01-28 Sprint

#15 - 01/15/2015 04:07 PM - Peter Amstutz

Comments on Python SDK @ 01/15/2015 10:08 am

I like the idea of tying together execution with iterables and futures, but there are some conceptual details that need to be carefully specified:

1. If data flow between tasks is intended to be controlled only by the start() process, that should be made explicit.
2. Is it allowable for WorkMethods to invoke (and wait on) other WorkMethods? If not, should this be explicitly disallowed?
3. What happens if a WorkMethod yields? Does the caller resolve it to a list? Error? Return multiple results some other way?
4. What happens when you pass an iterable to a WorkMethod? Does it always parallelize? What about when the user actually wanted to pass a literal list?
5. To tie these question together: can we handle situations where a WorkMethod produces N independent items of output, where we can immediately start processing a 2nd WorkMethod on each item before the first WorkMethod has actually completed?

Suggestion: we may need some type annotations on the WorkMethod parameters in order to disambiguate some of these situations.

(I should also mention, the common workflow language is dealing with these problems of implicit iteration, see https://groups.google.com/forum/#!topic/common-workflow-language/eGrNfpjuq2E and the discussion of "dot product", "cross product", and "port depth")

#16 - 01/15/2015 11:09 PM - Brett Smith

Peter Amstutz wrote:

> 1. If data flow between tasks is intended to be controlled only by the start() process, that should be made explicit.
> 2. Is it allowable for WorkMethods to invoke (and wait on) other WorkMethods? If not, should this be explicitly disallowed?

WorkMethods can invoke each other. The caller should not wait for the callee to actually execute, and the proposal provides users with no facilities to do this.

FutureOutput objects aren't futures, which is why I didn't just name the class Future. It holds information about work that has been scheduled, which a Dispatcher can use to schedule later work that needs the output. But they're not expected to have any user-facing methods, or other hooks for onComplete/onError-type operations. The caller can't do anything with it except pass it as an argument to other WorkMethods. If there's something in the FutureOutput section that isn't clear on this point, please let me know.

> 1. What happens if a WorkMethod yields? Does the caller resolve it to a list? Error? Return multiple results some other way?
> 2. What happens when you pass an iterable to a WorkMethod? Does it always parallelize? What about when the user actually wanted to pass a literal list?

Dispatchers need to be able to serialize all inputs and outputs to WorkMethods. If we support passing in an iterator, it'll have to become a list. Again, one WorkMethod calling another can't resolve the latter's output at all.

> 1. To tie these question together: can we handle situations where a WorkMethod produces N independent items of output, where we can immediately start processing a 2nd WorkMethod on each item before the first WorkMethod has actually completed?

Our first implementation won't be able to do this because Crunch doesn't currently support it, but a dispatcher with more smarts could. You'd just have to write the code so that the WorkMethod1 calls WorkMethod2 when it has an item it wants to process. The Dispatcher will schedule that work immediately. If the underlying system supports it, the proposal can handle the new work starting while WorkMethod1 is still running.

> Suggestion: we may need some type annotations on the WorkMethod parameters in order to disambiguate some of these situations.
>
> (I should also mention, the common workflow language is dealing with these problems of implicit iteration, see https://groups.google.com/forum/#!topic/common-workflow-language/eGrNfpjuq2E and the discussion of "dot product", "cross product", and "port depth")

The proposal does not include any implicit iteration. If you want to start a WorkMethod N times, you must call it N times. You can do this easily with a for loop, and I think that will be the most comfortable way for most of our users to achieve this.

#17 - 01/16/2015 09:14 PM - Tim Pierce

The notes on limiting concurrency remind me of one of the drums Peter's been beating for a while -- that Jobs and Tasks should properly be implemented as the same kind of object. If we could formally define a Task's resource requirements or inputs and outputs in a template, the way we currently do with Jobs, it would be easier for Crunch to introspect the Tasks to figure out which ones depend on each others' inputs, or what the resource constraints on concurrency will be, and so on.

#18 - 01/20/2015 06:58 PM - Peter Amstutz

Brett Smith wrote:

> FutureOutput objects aren't futures, which is why I didn't just name the class Future. It holds information about work that has been scheduled, which a Dispatcher can use to schedule later work that needs the output. But they're not expected to have any user-facing methods, or other hooks for onComplete/onError-type operations. The caller can't do anything with it except pass it as an argument to other WorkMethods. If there's something in the FutureOutput section that isn't clear on this point, please let me know.

The FutureOutput section isn't clear on this point and would benefit from inclusion of this discussion. Also, "FutureOutput" strongly implies it is a future in the CS sense. Consider renaming to something like "TaskDependency" or "TaskOutput".

>> 1. What happens if a WorkMethod yields? Does the caller resolve it to a list? Error? Return multiple results some other way?
>> 2. What happens when you pass an iterable to a WorkMethod? Does it always parallelize? What about when the user actually wanted to pass a literal list?
>
> Dispatchers need to be able to serialize all inputs and outputs to WorkMethods. If we support passing in an iterator, it'll have to become a list. Again, one WorkMethod calling another can't resolve the latter's output at all.

That's reasonable. Clarify this in the proposal.

>> 1. To tie these question together: can we handle situations where a WorkMethod produces N independent items of output, where we can immediately start processing a 2nd WorkMethod on each item before the first WorkMethod has actually completed?
>
> Our first implementation won't be able to do this because Crunch doesn't currently support it, but a dispatcher with more smarts could. You'd just have to write the code so that the WorkMethod1 calls WorkMethod2 when it has an item it wants to process. The Dispatcher will schedule that work immediately. If the underlying system supports it, the proposal can handle the new work starting while WorkMethod1 is still running.

I see. In that case, you build the dataflow directly as part of your program logic, instead than a set of components that are hooked together in the start() method.

> The proposal does not include any implicit iteration. If you want to start a WorkMethod N times, you must call it N times. You can do this easily with a for loop, and I think that will be the most comfortable way for most of our users to achieve this.

Now that I understand the proposed semantics for WorkMethods better, this is reasonable.

#19 - 01/22/2015 07:12 PM - Brett Smith

Tim Pierce wrote:

> The notes on limiting concurrency remind me of one of the drums Peter's been beating for a while -- that Jobs and Tasks should properly be implemented as the same kind of object. If we could formally define a Task's resource requirements or inputs and outputs in a template, the way we currently do with Jobs, it would be easier for Crunch to introspect the Tasks to figure out which ones depend on each others' inputs, or what the resource constraints on concurrency will be, and so on.

Right, and one of the goals of the SDK is to provide an API that's usable with both today's job/task dichotomy, and a future Crunch that flattens that hierarchy. In other words, Crunch scripts that stick to the advertised interfaces should transparently continue to work when this part of the Arvados API changes.

#20 - 01/22/2015 07:37 PM - Brett Smith

Peter Amstutz wrote:

> Brett Smith wrote:
>
>> FutureOutput objects aren't futures, which is why I didn't just name the class Future. It holds information about work that has been scheduled, which a Dispatcher can use to schedule later work that needs the output. But they're not expected to have any user-facing methods, or other hooks for onComplete/onError-type operations. The caller can't do anything with it except pass it as an argument to other WorkMethods. If there's something in the FutureOutput section that isn't clear on this point, please let me know.

The FutureOutput section isn't clear on this point and would benefit from inclusion of this discussion. Also, "FutureOutput" strongly implies it is a future in the CS sense. Consider renaming to something like "TaskDependency" or "TaskOutput".

Names with "Task" in them are undesirable because they're specific to current Crunch. Same goes for Job.

I feel like the English word "future" is too useful to be wholly owned by formal CS futures, especially given that our primary audience does not have a formal CS background. It tries to signal to the user: this object represents the output of another work unit, but you can't use it now, because it will only be available in the future. Imagine a user trying to use it like an object they instantiated and getting the exception message, AttributeError: 'FutureOutput' object has no attribute 'foobar'. That seems about as clear as we can expect a Python-generated error message to be, especially considering that the user may not know the class name since they never instantiate these objects directly themselves.

1. To tie these question together: can we handle situations where a WorkMethod produces N independent items of output, where we can immediately start processing a 2nd WorkMethod on each item before the first WorkMethod has actually completed?

Our first implementation won't be able to do this because Crunch doesn't currently support it, but a dispatcher with more smarts could. You'd just have to write the code so that the WorkMethod1 calls WorkMethod2 when it has an item it wants to process. The Dispatcher will schedule that work immediately. If the underlying system supports it, the proposal can handle the new work starting while WorkMethod1 is still running.

I see. In that case, you build the dataflow directly as part of your program logic, instead than a set of components that are hooked together in the start() method.

Yes. I mean, for simpler workflows, you can go either way, but if you want to do scheduling differently based on some aspect of the input, the way to do that is to have the method generating the input introspect it and do the scheduling dynamically. I have yet to puzzle out a way to lay this out at the start within the limitations of current Crunch.

I added a clarifying paragraph to the page for the points where you asked about that. Thanks.

#21 - 01/23/2015 10:57 PM - Tom Clegg

This looks really good.

It seems to me that Future is an appropriate term here: it's a placeholder for a value that isn't known yet. I think it's totally fine that it doesn't support all of the methods found in a generic future/promise interface.

I find the class name CrunchScript a little unsatisfying -- it seems weird to say you're "creating a CrunchScript" or "defining a CrunchScript" in the __main__ section of ... well, a crunch script -- although I'm not sure exactly what to change it to.

- CrunchAdapter would make sense when the script is an adapter between Crunch and some other program, but that's too specific.
- CrunchProcess suggests a UNIX process, which could only serve to confuse.
- CrunchJob is tempting, although it will become confusing if we rearrange Crunch such that jobs can run other jobs (instead of "jobs run tasks"), which we expect to do.
- CrunchWorker conflicts with (or corresponds nicely to?) our tendency to call compute nodes "workers" or "worker nodes".

The code in __main__, particularly the job_output_from mechanism, feels a bit awkward to me. I think I'm not quite following why it's the Dispatcher's job to know which work method yields the job output, even though the caller in __main__ also has to choose a CrunchScript method to call in order to make anything happen.

Instead of having a start method that always decides what to do, and a different mechanism for deciding what to save, could we just call the method whose output we want to save? It would change the examples a bit: the caller would invoke the start() method explicitly, which would return a generator or list of futures, and that list of futures would be the output. (This means start() wouldn't be a magic word: it could just as well be analyze_all_tumors() or whatever.)

Current:

-
  ```
  █ __name__ == '__main__':
      dispatcher = CrunchDispatcher(job_output_from=['analyze_tumors'])
      TumorAnalysis.run(dispatcher=dispatcher)
  ```

Suggestion 1:

-
  ```
  █ __name__ == '__main__':
      TumorAnalysis().start()
      # Or, if using non-default dispatcher:
      # TumorAnalysis(dispatcher=CustomDispatcher()).start()
  ```

- start()'s [future] return value would be used as the job output. Not sure how this magic would work, just expecting it's possible if we want it badly enough.

Suggestion 2:

-
  ```
  [redacted] __name__ == '__main__':
      TumorAnalysis().run('start')
  ```
  - (This is less magical: obviously the run() method knows that the return value of start() is supposed to be the job output.)

Suggestion 3:

-
  ```
  [redacted] __name__ == '__main__':
      CrunchJob.setOutput(TumorAnalysis().start())
  ```
  - (Less magical, but uses a native method call instead of passing a string with the name of the desired method, which is nice, IMO.)

#22 - 01/24/2015 05:00 AM - Brett Smith

Tom Clegg wrote:

> I find the class name CrunchScript a little unsatisfying -- it seems weird to say you're "creating a CrunchScript" or "defining a CrunchScript" in the __main__ section of ... well, a crunch script -- although I'm not sure exactly what to change it to.

Pro of the name: it really drives home the point that all your code should go in the class.

Another downside of the name, though: this API will let you inherit from other CrunchScripts in interesting ways. Like, you might have a CrunchPicardMixin that provides WorkMethods for common Picard calls, and you can just inherit from that to get those and use them in conjunction with custom WorkMethods in different Crunch scripts. (We already have several examples of this happening in the wild, except right now it's much less organized since every Crunch script expects to be integrated different ways.) The CrunchScript name obscures this.

More canoodling:

- CrunchWork
- ArvadosWork
- ArvadosJob (has the same downsides as CrunchJob you mentioned, though)
- ArvadosCompute

> The code in __main__, particularly the job_output_from mechanism, feels a bit awkward to me. I think I'm not quite following why it's the Dispatcher's job to know which work method yields the job output, even though the caller in __main__ also has to choose a CrunchScript method to call in order to make anything happen.

I feel like there's a misunderstanding here, although maybe we're just using different lingo?

The __main__ block doesn't choose a *method* to run. All it *has* to do is call UserCrunchScript.run(). (If you consider picking the UserCrunchScript plus run() method pair to be choosing a method, well, then my comments might be moot.) run() is implemented in the SDK's CrunchScript superclass. It sets up a dispatcher (if the user didn't provide one), then coordinates with that to figure out which WorkMethod is actually supposed to be run in this invocation, deserialize its parameters, and invoke the user's code.

In other words, for the first SDK implementation, think of CrunchScript.run() as handling all the work that gets done in boilerplate at the top of today's Crunch script: loading the current job and task, turning parameters into real objects, etc. Most of the real work there will be done in the Dispatcher, but run() will build the bridge between them.

The Dispatcher has to be the place to say "these tasks provide the job's output," because the Crunch Dispatcher is the only piece of code that knows anything about tasks. Imagine it from the future: if we lived in a world without tasks, and only jobs, it wouldn't make sense to name methods that provide the job's output—because each method would be a job, and provide its own output. (I think? Maybe I misunderstand the general shape of future plans.)

Maybe a useful analogy: CrunchScript is to the Dispatchers as ActiveRecord is to specific database drivers.

> Instead of having a start method that always decides what to do

Put in today's language, start() is the code for task 0. It doesn't run every time the Crunch script is called, just on the first time the script is run for the job. (Hmm, in retrospect, this might be too specific to current Crunch. I'm honestly not sure—will other schedulers need/want an idea of a default entry point?)

#23 - 01/24/2015 09:42 AM - Tom Clegg

- Category set to Crunch

How about CrunchModule? Too vague?

> The *main* block doesn't choose a method to run.

I was just referring to the TumorAnalysis.run() method. And I'm wondering about rephrasing it so "the output of the job is the output promised by analyze_tumors()" is expressed by *calling* analyze_tumors(). The @WorkMethod() decorator could be responsible for all the magic that currently lives in run(), right?

start() starts to look like just another WorkMethod if

- it gets @WorkMethod() decoration,
- it adds the word return before self.analyze_tumors(...), and
- the __main__ section tells Crunch (using whatever syntax) that start() is the work method whose output is the job output.

In the current TumorAnalysis example, FutureOutput objects are seen by start(), and passed to work methods. None of the methods actually *return* FutureOutputs, and the work methods themselves never see them at all: the only place FutureOutputs come from is the behind-the-scenes WorkMethod magic. So I'm guessing you're only ever expecting to see FutureOutput objects in start() (task 0). But I think it would be even better to generalize this by letting work methods return futures:

```
class TumorAnalysis(CrunchScript):
    @WorkMethod()
    def classify_then_analyze(self, input):
        # This runs in task 0 (assuming __main__ looks like it does
        # below, at least). It returns a FutureOutput, which means
        # more tasks need to be queued: one for the returned future,
        # plus one for each of that future's N inputs/dependencies.
        return self.analyze_tumors(self.classify(in_file)
                                   for in_file in input.all_files()
                                   if in_file.name.endswith('.fastj'))

    @WorkMethod()
    def classify(self, in_file):
        # This runs in each of tasks 1..N. It returns a collection or None.
        ...
        if ...:
            ...
            return coll
        return None

    @WorkMethod()
    def analyze_tumors(self, results):
        # This runs in task N+1. The dispatcher runs this task upon noticing
        # that all of the future outputs passed in as arguments have been
        # resolved. It returns a stringifiable object, which gets saved and
        # -- because returning here resolves a promise that was returned
        # by the work method called by __main__ in task 0 -- becomes the
        # output of the job. The job is complete.
        ...
        return compiled

if __name__ == '__main__':
    cruncher = TumorAnalysis(dispatcher=CrunchDispatcher())
    futureOut = cruncher.classify_then_analyze()
    whateverYouDoToSetOutputHere(futureOut)
```

This would make it possible for TumorAnalysis's work methods to call SomeOtherUpstreamAnalysis's work methods without knowing whether those upstream methods involve their own little dependency trees of subtasks.

...in other words, write pipelines in Python, using exactly the same facilities you use when writing the individual components/modules/jobs/scripts/whatever-we-call-them. (Hand-waving away, for the moment, questions like how to import CrunchWhatever classes from one repository to another.)

#24 - 01/24/2015 02:51 PM - Brett Smith

- Category deleted (Crunch)

Tom Clegg wrote:

> How about CrunchModule? Too vague?

Only slightly more than the others. I could go for it.

>> The *main* block doesn't choose a method to run.

> I was just referring to the TumorAnalysis.run() method. And I'm wondering about rephrasing it so "the output of the job is the output promised by analyze_tumors()" is expressed by *calling* analyze_tumors().

Ah, okay. I think I understand better where you're going now, and it makes sense. And it would work fine for the example scripts, but I think it would impose some awkward limitations on more serious Crunch scripts.

- What if your workflow forks, and you want your final output to include both ends of the fork? For example, say you preprocess the data a bit, and then feed those results to two related but different analysis tasks. Of course, you could write a little task that ties them together, but then we run into the issue that
- This works easily if you can dispatch all your final output tasks from one place like the start() method in the examples, but I think it's trickier when you start having WorkMethods calling each other to dispatch work more dynamically. And the SDK expects that you'll write your code this way if you want that kind of dynamism for performance. See my discussion with Peter around note-16.

I think all of these issues are surmountable with code, but I think the SDK can save you the hassle of writing that code yourself if you declare to it, "These are the WorkMethods that produce interesting results."

What if, instead of giving this information to the Crunch dispatcher, CrunchModule recognized a class attribute that listed the names of WorkMethods that generate "useful" (as opposed to intermediate) output? The first implementation Crunch dispatcher would use this information to set the job output from task outputs, but I think the concept is general enough that we could continue to support it as we get new dispatchers.

> In the current TumorAnalysis example, FutureOutput objects are seen by start(), and passed to work methods. None of the methods actually *return* FutureOutputs, and the work methods themselves never see them at all: the only place FutureOutputs come from is the behind-the-scenes WorkMethod magic. So I'm guessing you're only ever expecting to see FutureOutput objects in start() (task 0).

I am not. Again, see note 16.

#25 - 01/24/2015 03:00 PM - Brett Smith

- Category set to Crunch

#26 - 01/25/2015 12:25 AM - Tom Clegg

Here's a strawman for "how modules call one another's methods":

```
class ModuleA(CrunchModule):
    @WorkMethod()
    def splitAndMerge(self, input):
        return self.merge(self.split(input))

    @WorkMethod()
    def split(self, input):
        ret = []
        for f in input.all_files():
            for chr in ['22', 'X', 'Y']:
                # (make a new collection with just the selected bits)
                ret.append(newcoll)
        return ret

    @WorkMethod()
    def merge(self, inputs):
        coll = Collection()
        for chr in ['22', 'X', 'Y']:
            for i in inputs:
                # (...)
        return coll

class ModuleB(CrunchModule):
    @WorkMethod()
    def downstreamAnalysis(self, input):
        preprocessed = ModuleA(dispatcher=self.dispatcher).splitAndMerge(input)
        return self.postprocess(preprocessed)

    @WorkMethod()
    def postprocess(self, input):
        # (real work happens here)
        return stuff
```

It seems to me the question of whether ModuleA's splitAndMerge() is an "output method" or an "intermediate method" depends on whether it was called by a __main__ (crunch_script=ModuleA.py) or by ModuleB's downstreamAnalysis().

> What if your workflow forks, and you want your final output to include both ends of the fork?

I figured you'd express this either by returning a list of the FutureOutputs obtained by invoking the forks (which we would presumably take to mean "concatenate these") or by passing those FutureOutputs to a work method that knows how to merge them (if you wanted something other than concatenation, or if you wanted to blurt some log messages, or something like that).

One feature of doing it explicitly (either by returning a list, or by making your own merge method), rather than just telling dispatcher which method is interesting, is that it's easy to see (and alter) how the order gets decided when concatenating the return values from multiple invocations of the output method(s).

But the main reason I like the "call the method whose output you want" idea is that it looks more like normal programming:

- If work method "Foo" is public, outsiders can call it. If it isn't public, outsiders don't need to know it even exists.
- Output/result is returned using "return". The caller decides (upon receipt) what to do with it.

> ...if you declare to it, "These are the WorkMethods that produce interesting results."

Isn't that what my proposed __main__ already does by *calling* the method(s) whose outputs it's interested in? If there are work methods whose output isn't interesting, and {whatever __main__ calls} doesn't invoke them, then there shouldn't be any need to run them at all.

At any rate, all I'm trying to do here is take the FutureObject concept all the way through to __main__, which seems to come down to offering an alternate way of specifying an entry point for "task 0". There doesn't seem to be anything fundamentally incompatible about it, so it shouldn't be holding up progress on implementing the way you've already planned.

Sort of tangentially, here's an interesting post about promises that I wish I had read before writing the Workbench uploader.

#27 - 01/25/2015 03:05 PM - Brett Smith

Let me talk about implementation concerns for a minute, because I think that's my main source of worry about the idea. I apologize if you're already aware of a lot of this stuff; I just want to be very sure we're on the same page.

if __name__ == '__main__' is pure Python; the SDK does nothing to make that go. The condition is true if this code is the "program" being run, rather than being imported from other code. If the user wanted to issue the appropriate commands (whatever they are) for the CrunchModule to start doing work right after the class definition, without this guard, we can't stop them—and it will still work fine when this file is the Crunch script. Using the condition lets other users can import this file to extend the CrunchModule, without worrying that doing so will immediately kick off Crunch work.

When this file actually is the Crunch script named in a job, then the code under this condition will run for every task. That's how Crunch works: it runs the exact same script over and over again, with different environment variables to provide input.

Every serious Crunch script starts out by loading the current task to get its own parameters, turning those parameters into real objects, and then running the code for "this task," whatever that means. The SDK can do all of that for the user, including running the code for "this task" by calling a WorkMethod named in a special task parameter. But to do that, the user needs to hand over control to the SDK almost immediately, before we've even figured out whether this is task 0 or task N.

In my proposal, CrunchModule.run() is that hand-off. Your proposal has no explicit hand-off point. Implementing it would require much deeper magic in WorkMethod; something along the lines of:

- When the first WorkMethod is called, load the current task.
- If this is task 0, then run the WorkMethod's user code normally, returning the result.
- Otherwise, ignore the user's call completely. Figure out which WorkMethod this task is meant to run, and run it. After we're done, prevent future WorkMethod calls from doing anything—don't run user code, don't dispatch tasks—because they're coming from the job output definition inside the __main__ block, and we already ran all that during task 0, so doing it again is redundant.

And that's just the implementation for current Crunch. Never mind adapting it to other schedulers.

While I hear what you're saying about your version looking more natural, users are going to have to give the SDK *some* space to let it help them. I think an explicit hand-off in CrunchModule.run() is better than implicit one, both for us and our users. Us, because it's much simpler to implement and maintain. Our users, because it helps make the rules clearer. With your approach, there will probably almost always be something someone can do in the __main__ block that will mess things up, because whatever executes there is completely outside the SDK's control or introspection. It seems easier for them to learn and remember "call CrunchMethod.run()" than "define your job's output flow in your __main__ block, using/avoiding the following structures/tools/whatever."

Plus, you can import this, and it will tell you very plainly: "Explicit is better than implicit."

Tom Clegg wrote:

> Here's a strawman for "how modules call one another's methods":

I was envisioning extending via inheritance, rather than instantiating multiple CrunchModules. Right now I think I can sustain both approaches; but do you feel especially strongly that one or more specific models *must* work?

#28 - 01/25/2015 09:18 PM - Tom Clegg

Brett Smith wrote:

> When this file actually is the Crunch script named in a job, then the code under this condition will run for every task. That's how Crunch works: it runs the exact same script over and over again, with different environment variables to provide input.

Yes, that happens behind the scenes. I wasn't sure we wanted script authors to think of __main__ that way. But I agree there's a line somewhere between conveniently-magic and what-is-this-i-dont-even, and I can accept that co-opting __main__ (or invoking task 0 differently than task N>0 so __name__ *isn't* '__main__') is on the wrong side of it.

> With your approach, there will probably almost always be something someone can do in the __main__ block that will mess things up, because whatever executes there is completely outside the SDK's control or introspection.

I think the question of what to put in __main__ is confusing in that we're putting a bunch of magic in CrunchModule to make it feel like all the work methods run in the same process -- then expecting it to be obvious to the user that __main__ runs again every time a WorkMethod is invoked.

(Neither approach really prevents __main__ from messing things up, except by not looking like a reasonable place to put some code. Right?)

> It seems easier for them to learn and remember "call CrunchMethod.run()" than "define your job's output flow in your __main__ block, using/avoiding the following structures/tools/whatever."

Both approaches define the job's output flow in __main__, they just use different syntax. And both require that you avoid a lot of things. But I see your point now (right?) that the more __main__ looks like a job-bootstrapping program, the more tempting it will be to do programming there that doesn't work the way you expect. (And making it actually *work* that way is either too magical to understand or unreasonably icky/impossible to implement.)

I still think the way "the output method" is defined is weird (although I do see why it's easier to implement).

What if we define "the output" by returning it from start()? That way, we only ask the question once, instead of expecting __main__ to tell the dispatcher the same job_output_from during each task. And we can have ModuleB call ModuleA's start() method ("do your thing") and use the result as an input to something else (or as its own output), without knowing how ModuleA named its output methods.

> Plus, you can import this, and it will tell you very plainly: "Explicit is better than implicit."

Sure, OTOH our goal here is to *avoid* being explicit (in Crunch scripts) about the mechanics of scheduling tasks, passing information between them, resuming program flow at the appropriate place when futures are fulfilled, etc.

> I was envisioning extending via inheritance, rather than instantiating multiple CrunchModules. Right now I think I can sustain both approaches; but do you feel especially strongly that one or more specific models *must* work?

To me, the relationship between those two modules (which I think is representative of common/important cases) seems more like "ModuleB uses ModuleA" than "ModuleB is a special kind of ModuleA", which generally means "instantiate and use" fits better than "inherit from".

Using *multiple* inheritance this way seems to descend quickly to madness. Consider a module "D" that knows about two different modules, "E" and "F", which do similar things. "D" uses the input filename to decide which to invoke. "E" and "F" happen to have some methods with the same name (unsurprising, since they do similar things). If "D" uses "E" and "F" by inheriting from them, we get stuff like this:

```
#!/usr/bin/python

class E(object):
    def allgood(self):
        print "I expect E. I get",
        print self.bar()
    def bar(self):
        return "E"

class F(object):
    def uhoh(self):
        print "I expect F. I get",
        print self.bar()
    def bar(self):
        return "F"

class D(E, F):
    def run(self):
        self.allgood()
        self.uhoh()

D().run()
"""
I expect E. I get E
I expect F. I get E
"""
```

#29 - 01/26/2015 01:35 AM - Brett Smith

Tom Clegg wrote:

> I think the question of what to put in __main__ is confusing in that we're putting a bunch of magic in CrunchModule to make it feel like all the work methods run in the same process -- then expecting it to be obvious to the user that __main__ runs again every time a WorkMethod is invoked.

I think this is going to be clearer to users than you imagine. Because it's going to put a lot of restrictions on the way the program is written. You can't share data by mutating globals or instance state. You can't pass objects to WorkMethods that the SDK doesn't know how to serialize. And so on.

(I'm hoping I can help provide users with some guide rails to spare them long debugging sessions. For example, maybe raise an exception if a WorkMethod assigns an instance variable—"Other WorkMethods won't see this change. Pass the data around as an argument instead.")

Because of that, I've tried to design the API so that it's always explicit when magic is happening. Hence you decorate WorkMethods yourself, instead of CrunchModule wrapping public instance methods automatically. And you call CrunchMethod.run() to have the SDK figure out which of your methods is supposed to execute this time around, and run that. And I hope *what* the SDK does can be comprehensible to users ("when I call a WorkMethod, it creates a task to run my code later"), even if they don't understand how it's implemented.

Conversely: there is no explicit magic in if __name__ == '__main__', and I would rather not add any implicit magic.

> I still think the way "the output method" is defined is weird (although I do see why it's easier to implement).

This I can easily agree with.

> What if we define "the output" by returning it from start()?

Then I wonder about the interaction with note-16 that I brought up earlier: if start() calls a few WorkMethods that call a few other WorkMethods that call yet different still WorkMethods that produce the final output, how do you express that those final WorkMethods produce the real output? Note that you won't have FutureOutput objects from the final batch of methods, because they won't be invoked in the process that runs start().

> To me, the relationship between those two modules (which I think is representative of common/important cases) seems more like "ModuleB uses ModuleA" than "ModuleB is a special kind of ModuleA", which generally means "instantiate and use" fits better than "inherit from".

I agree philosophically, but again, implementation concerns: this requires the Dispatcher to be able to find WorkMethods at potentially arbitrary locations, rather than attached directly to the current CrunchModule. I'll play a little to see how involved it would be to resolve this.

> Using *multiple* inheritance this way seems to descend quickly to madness.

I agree that the problems you're worried about are real and directly attributable to multiple inheritance, but I think they're unlikely to be a major concern in our use cases. I think inheriting from multiple CrunchModules will be rare to begin with, and when it occurs I think there are unlikely to be name conflicts between WorkMethods that require caring about the method resolution order. In the rare case it does happen, it's relatively straightforward to write a method that expressly calls the one you're interested in.

#30 - 01/26/2015 04:25 PM - Tom Clegg

Brett Smith wrote:

> maybe raise an exception if a WorkMethod assigns an instance variable—"Other WorkMethods won't see this change. Pass the data around as an argument instead."

Could we make the WorkMethods be class methods instead of instance methods?

> And you call CrunchMethod.run() to have the SDK figure out which of your methods is supposed to execute this time around, and run that.

WDYT of making the entry point something like dispatcher.runTask() instead? That might be more suggestive that

- __main__ invokes each task
- you're handing off to the SDK now, and *it* will invoke your CrunchMethod.

Of course this would mean telling the dispatcher about your CrunchMethod instead of the other way around (which also seems a bit more natural to me: "hey dispatcher, run this" instead of "hey crunchmethod, ask this dispatcher to run you".

```
dispatcher = Dispatcher(CrunchMethod(), job_output_from=['analyze_tumors'])
dispatcher.runTask()
```

> > What if we define "the output" by returning it from start()?
>
> Then I wonder about the interaction with note-16 that I brought up earlier: if start() calls a few WorkMethods that call a few other WorkMethods that call yet different still WorkMethods that produce the final output, how do you express that those final WorkMethods produce the real output? Note that you won't have FutureOutput objects from the final batch of methods, because they won't be invoked in the process that runs start().

Isn't this the same problem as: what if the output method calls another work method, which calls another work method? How about this:

```
class CrunchScript:
    def _start(self, input):
        # Invoked by dispatcher for task 0.
        self.output(self.start(input))

    @WorkMethod()
    def output(self, the_output):
        # This is always considered an output method, even if not explicitly listed.
        return the_output
```

> > To me, the relationship between those two modules (which I think is representative of common/important cases) seems more like "ModuleB uses ModuleA" than "ModuleB is a special kind of ModuleA", which generally means "instantiate and use" fits better than "inherit from".
>
> I agree philosophically, but again, implementation concerns: this requires the Dispatcher to be able to find WorkMethods at potentially arbitrary locations, rather than attached directly to the current CrunchModule. I'll play a little to see how involved it would be to resolve this.

Serialize with class name?

> > Using *multiple* inheritance this way seems to descend quickly to madness.
>
> I agree that the problems you're worried about are real and directly attributable to multiple inheritance, but I think they're unlikely to be a major concern in our use cases. I think inheriting from multiple CrunchModules will be rare to begin with, and when it occurs I think there are unlikely to be name conflicts between WorkMethods that require caring about the method resolution order. In the rare case it does happen, it's relatively straightforward to write a method that expressly calls the one you're interested in.

I don't think this will be rare. E.g., I have a CrunchModule and a slight modification (subclass!) of that CrunchModule. I choose one depending on the input -- or I run both modules on every input, so I can diff them.

Dare I ask, what is the relatively straightforward solution? (Was the margin too small to contain it?)

#31 - 01/26/2015 10:51 PM - Brett Smith

Tom Clegg wrote:

> Brett Smith wrote:
>
> > maybe raise an exception if a WorkMethod assigns an instance variable—"Other WorkMethods won't see this change. Pass the data around as an argument instead."
>
> Could we make the WorkMethods be class methods instead of instance methods?

This just shifts the problem around, since classes can have arbitrary attributes attached to them as well, and the same issues exist there.

> > And you call CrunchMethod.run() to have the SDK figure out which of your methods is supposed to execute this time around, and run that.
>
> WDYT of making the entry point something like dispatcher.runTask() instead? That might be more suggestive that
>
> - __main__ invokes each task
> - you're handing off to the SDK now, and *it* will invoke your CrunchMethod.
>
> Of course this would mean telling the dispatcher about your CrunchMethod instead of the other way around (which also seems a bit more natural to me: "hey dispatcher, run this" instead of "hey crunchmethod, ask this dispatcher to run you".

Yeah, this is probably an improvement. My only concern is the name runTask, since "task" is Crunch-specific. run_work() would have a nice symmetry with WorkMethod.

> > Then I wonder about the interaction with note-16 that I brought up earlier: if start() calls a few WorkMethods that call a few other WorkMethods that call yet different still WorkMethods that produce the final output, how do you express that those final WorkMethods produce the real output? Note that you won't have FutureOutput objects from the final batch of methods, because they won't be invoked in the process that runs start().
>
> Isn't this the same problem as: what if the output method calls another work method, which calls another work method?

Well, yes, but the approach of listing WorkMethods that generate output is intended to neatly sidestep this problem: you list the method at the *end* of that chain, instead of the one at the front (or you list them both, depending on exactly what you want). The final output will be generated by combining those, just like Crunch does now with all tasks.

> How about this:

I'm not 100% sure if you intend "start" in this example to be a new WorkMethod or not. But either way: as the proposal is currently written, this would cause the SDK to schedule an "output" task at sequence N+1, where N is the sequence number that "start" executes with (whether that's 0 or 1). But in the scenario I described, N+1 is too soon: the other tasks at sequence N+1 will schedule tasks at sequence N+2, and those will schedule tasks at sequence N+3, and *those* generate the output you're interested in. You won't be able to get what you want running in sequence N+1.

This is the disconnect I probably spent the most time struggling with in the design: there is zero correlation between the time a task finishes, and the time that the output you care about is available. Because that's effectively an implementation detail, depending on whether and how the task you start schedules subtasks to generate the interesting output. And you can't know that ahead of time, because that's the halting problem. As far as I can tell, you need an out-of-band mechanism to say what tasks generate the interesting output, because there's no reliable way to express your desires at task scheduling time.

> > I agree philosophically, but again, implementation concerns: this requires the Dispatcher to be able to find WorkMethods at potentially arbitrary locations, rather than attached directly to the current CrunchModule. I'll play a little to see how involved it would be to resolve this.
>
> Serialize with class name?

It's going to be a little more involved than that; we'll have to serialize the module name too, since class names don't have to be unique across namespaces, and then we may need to dynamically import that module (in case the user did the original import directly in the WorkMethod). But if you're okay with the SDK running that much code automatically, I guess it'll work.

> > I agree that the problems you're worried about are real and directly attributable to multiple inheritance, but I think they're unlikely to be a major concern in our use cases. I think inheriting from multiple CrunchModules will be rare to begin with, and when it occurs I think there are unlikely to be name conflicts between WorkMethods that require caring about the method resolution order. In the rare case it does happen, it's relatively straightforward to write a method that expressly calls the one you're interested in.
>
> I don't think this will be rare. E.g., I have a CrunchModule and a slight modification (subclass!) of that CrunchModule. I choose one depending on the input -- or I run both modules on every input, so I can diff them.

Shouldn't you just submit this as two jobs (or three, the two source processes plus the diff)?

I don't think this is entirely a rhetorical question. The SDK has the goal of making it easy to use Arvados effectively. If certain patterns are hard to implement because you should deal with those at other layers of Arvados (e.g., the pipeline layer), that actually seems like a positive to me.

> Dare I ask, what is the relatively straightforward solution? (Was the margin too small to contain it?)

My memory of your example drifted over the course of this thread. From a subclass, calling a specific superclass' method is easy by changing the first argument to super(). Resolving your example is not simple in the general case.

#32 - 01/27/2015 02:21 PM - Brett Smith

Brett Smith wrote:

> This is the disconnect I probably spent the most time struggling with in the design: there is zero correlation between the time a task finishes, and the time that the output you care about is available. Because that's effectively an implementation detail, depending on whether and how the task you start schedules subtasks to generate the interesting output. And you can't know that ahead of time, because that's the halting problem. As far as I can tell, you need an out-of-band mechanism to say what tasks generate the interesting output, because there's no reliable way to express your desires at task scheduling time.

Maybe a clearer way to put this is: when you schedule a task, the SDK can't know whether that task will generate useful output after it runs, or schedule more tasks to do that. Therefore, it can't know whether you *really* want follow-up tasks to be scheduled at sequence N+1 (the task generates output directly), or N+M (there are more layers of tasks between what you schedule and the final output).

If we could figure out a general solution to that general problem, I think it would be helpful for all kinds of reasons, including avoiding out-of-band lists of output tasks. But it bears a strong resemblance to the halting problem, so I'm skeptical it's doable.

#33 - 01/27/2015 02:54 PM - Brett Smith

But, if we can't schedule things perfectly up-front, we can patch things over after-the-fact. Here's one idea:

Permit WorkMethods to return a FutureOutput, or a list of FutureOutputs. This means, "My output is the output of these task(s)." The SDK records this information somewhere in the starting task record (TODO: figure out where).

All the details about scheduling called WorkMethods remain the same.

When the SDK starts a new task, one of its existing responsibilities is to fill in real objects where FutureOutputs were used before. With the change above, during this process, the task referred to by the FutureOutput at call time may now say, "my real output is the output of these task(s)." If those tasks are also done, great, we can continue with deserialization as normal. If they're not, however, the SDK "reschedules" the task: it clones the task that it's supposed to be running with a sequence number max(sequence numbers of the referenced tasks)+1, and stops without running the user code associated with the WorkMethod.

TODO: When a WorkMethod returns FutureOutputs, should the SDK schedule a task to
set the task's real output after the tasks referenced by the FutureOutputs are done? Or should collective outputs just be built on the fly as needed?

Pros of this approach: I think it's a pretty clear extension of the SDK's existing ideas. It's basically no extra work for the user.

Cons of this approach: It's very unclear how adaptable these rules will be to schedulers besides Crunch. It's potentially very high overhead on the compute nodes: it's not difficult to imagine scenarios where an entire suite of tasks at sequence N that are waiting for later output, so there's tons of dispatch overhead just rescheduling tasks. I think the performance hit of this could be surprising to users who don't understand how the SDK is implemented under the hood.

I don't really like this idea, but maybe it's useful as a starting point to get us somewhere good.

#34 - 01/29/2015 04:41 PM - Ward Vandewege

- Status changed from New to In Progress

#35 - 01/29/2015 07:41 PM - Tom Clegg

- Target version changed from 2015-01-28 Sprint to 2015-02-18 sprint

#36 - 01/29/2015 07:42 PM - Tom Clegg

- Story points changed from 3.0 to 1.0

#37 - 02/17/2015 07:45 PM - Brett Smith

I've updated the wiki page with our current thinking about the SDK API, mostly focused on setting job/task output, with the extension to support returning FutureOutput objects. Tom suggested a way to do this that doesn't incur quite as much overhead as I feared earlier, so it seems doable.

#38 - 02/18/2015 08:12 PM - Brett Smith

- Target version changed from 2015-02-18 sprint to 2015-03-11 sprint

#39 - 02/27/2015 02:57 PM - Peter Amstutz

Something to look at for inspiration:

http://swift-lang.org/main/

The gist is it Swift is a "parallel scripting language" where you write code that looks roughly like a conventional scripting language, but variables are single-assignment/immutable (I think), all steps run concurrently, and it does dependency analysis to actually schedule the correct sequencing of tasks. I think if we could approximate some of those features within a Python-based DSL/metaprogramming that would be very powerful (and this seems to be the direction that the design is already going towards.)

#40 - 02/27/2015 03:27 PM - Peter Amstutz

What happens if a user who is not so clever with Python does something like this? (I think it does the right thing, but I just want to be clear about it)

```
results = []
for in_file in input.all_files():
  if in_file.name.endswith('.fastj'))
    results.append(self.classify(in_file))
return self.analyze_tumors(results)
```

How deeply into the "results" data structure does it look for Future objects? Is it restricted to "dict" and "list" so they can be traversed reliably without reflection?

#41 - 03/11/2015 07:08 PM - Brett Smith

- Target version changed from 2015-03-11 sprint to 2015-04-01 sprint

#42 - 04/01/2015 07:08 PM - Tom Clegg

- Target version changed from 2015-04-01 sprint to 2015-04-29 sprint

#43 - 04/29/2015 07:08 PM - Tom Clegg

- Target version changed from 2015-04-29 sprint to 2015-05-20 sprint

#44 - 04/29/2015 07:27 PM - Brett Smith

- Target version deleted (2015-05-20 sprint)

#45 - 04/29/2015 07:30 PM - Ward Vandewege

- Target version set to 2015-05-20 sprint

#46 - 05/01/2015 01:31 PM - Ward Vandewege

- Target version changed from 2015-05-20 sprint to Arvados Future Sprints

#47 - 11/09/2015 03:58 AM - Brett Smith

- Target version changed from Arvados Future Sprints to Deferred

#48 - 10/21/2016 12:49 AM - Tom Morris

- Assigned To changed from Brett Smith to Tom Morris

#49 - 12/13/2016 07:52 PM - Tom Morris

- Status changed from In Progress to Closed

- Assigned To deleted (Tom Morris)

- Target version deleted (Deferred)