# Building workflows with Spotify's


Workshop for e-Infrastructures for Massively Parallel Sequencing
Uppsala, Jan 19-20, 2015

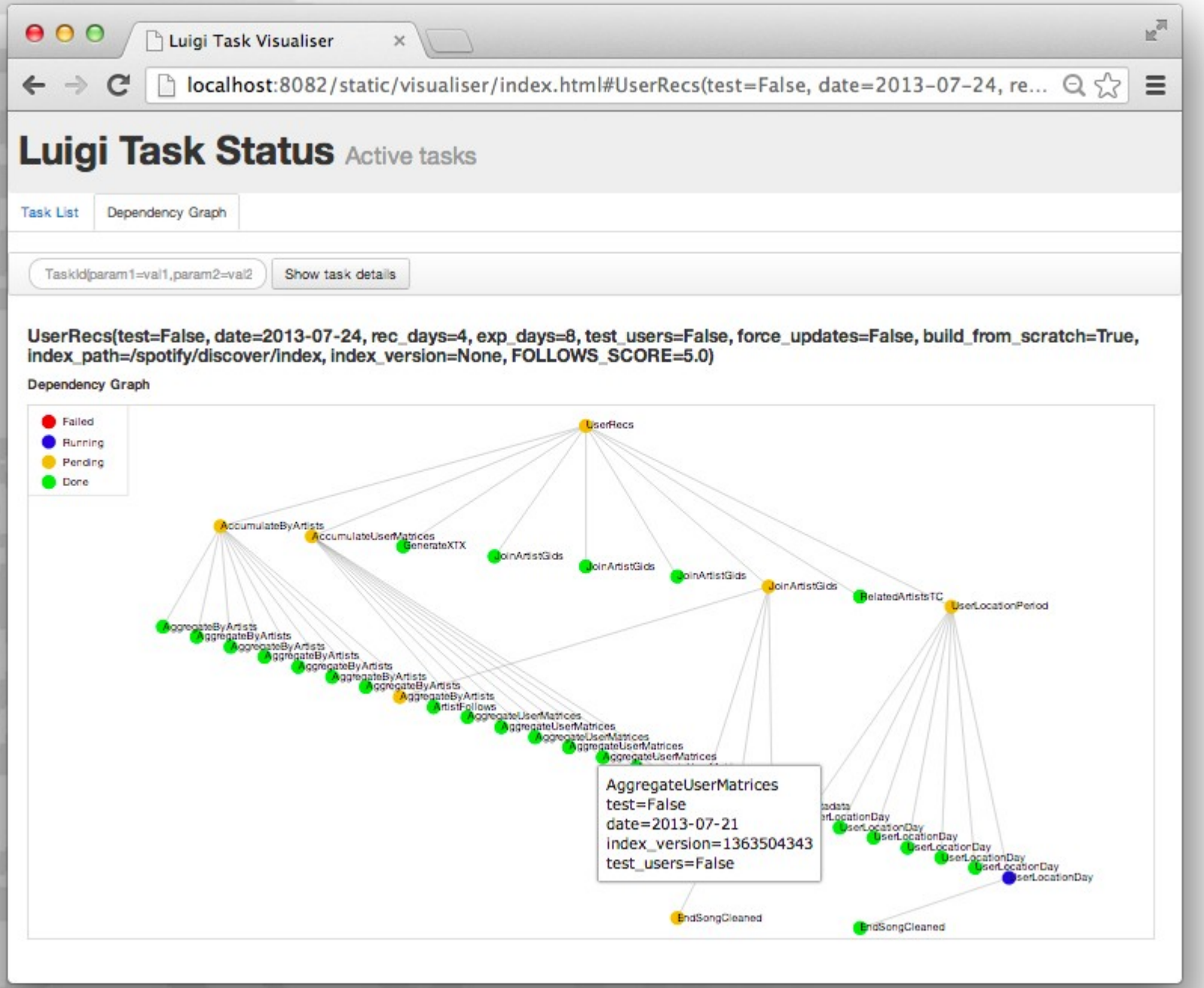
Samuel Lampa
UU/Dept of Pharm. Biosci. & BILS
samuel.lampa@bils.se
samuel.lampa.co

UPPSALA
UNIVERSITET

# Luigi Short Facts

- **Batch** (No streaming)
- Both **Command line** and **Hadoop** execution
- Does not replace Hive, Pig, Cascading etc
  - instead used to stitch many such jobs together
- Powers 1000:s of jobs every day at Spotify
- Communication / synchronization between tasks via “targets” (Normal file, HDFS, database …)
- Dependencies hard coded in tasks
- **Pull based** (ask for task, and it figures out dependencies)
- Main features: Scheduling, Dependency Graph,
  Basic multi-node support (via central planner daemon)
- github.com/spotify/luigi
- Easy to install: **pip install luigi [tornado]**

UPPSALA
UNIVERSITET



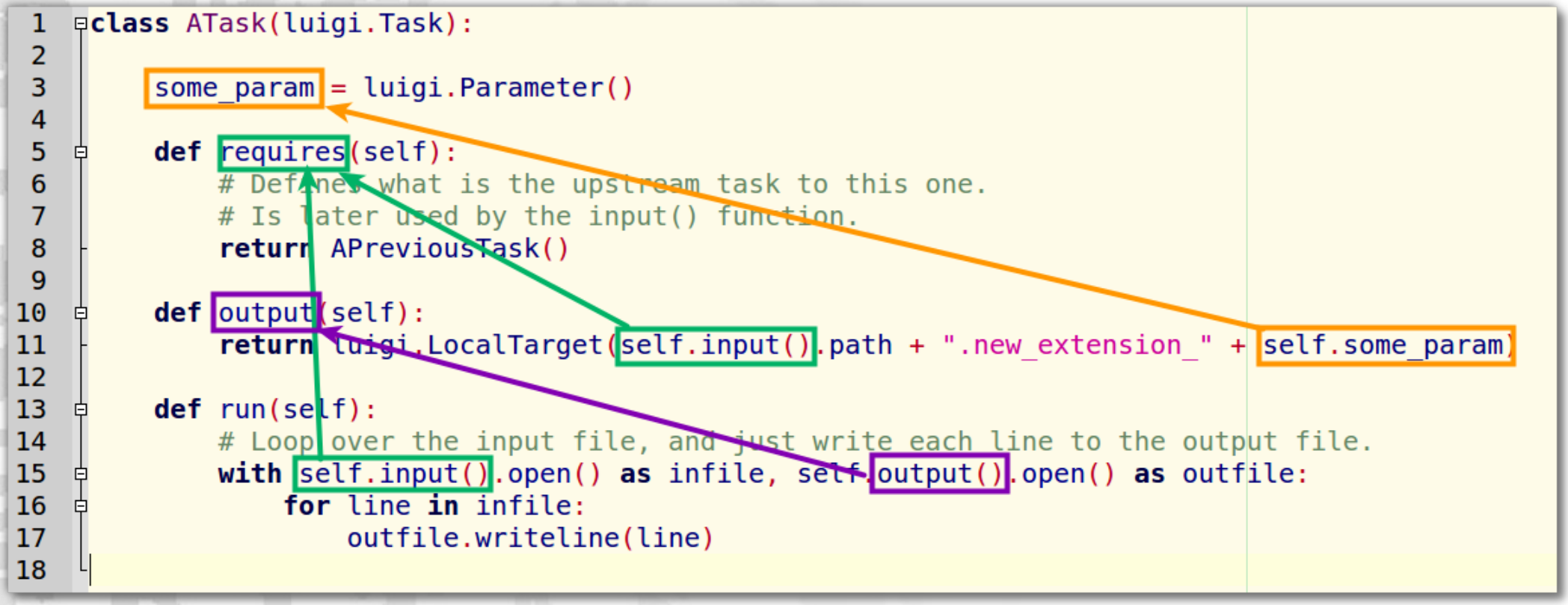

UPPSALA UNIVERSITET

# A Basic Luigi Task

```python
class ATask(luigi.Task):

    some_param = luigi.Parameter()

    def requires(self):
        # Defines what is the upstream task to this one.
        # Is later used by the input() function.
        return APreviousTask()

    def output(self):
        return luigi.LocalTarget(self.input().path + ".new_extension_" + self.some_param)

    def run(self):
        # Loop over the input file, and just write each line to the output file.
        with self.input().open() as infile, self.output().open() as outfile:
            for line in infile:
                outfile.writeline(line)

```

UPPSALA
UNIVERSITET

# A Basic Luigi Task



# Calling tasks from the commandline

```
$ python mytasks.py --local-scheduler ATask --some-param SomeValue
```

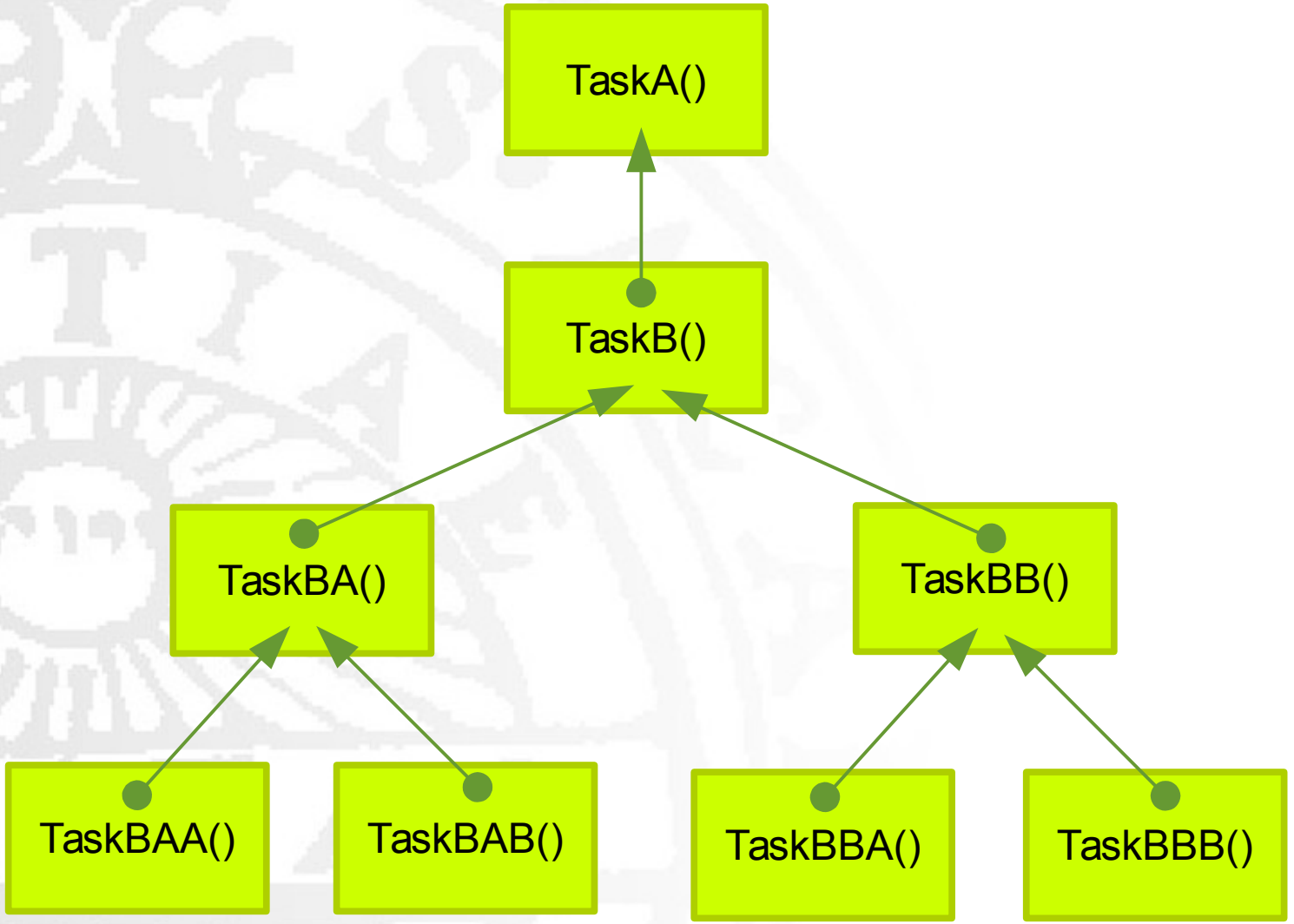
UPPSALA
UNIVERSITET

# Defining workflows: Default way

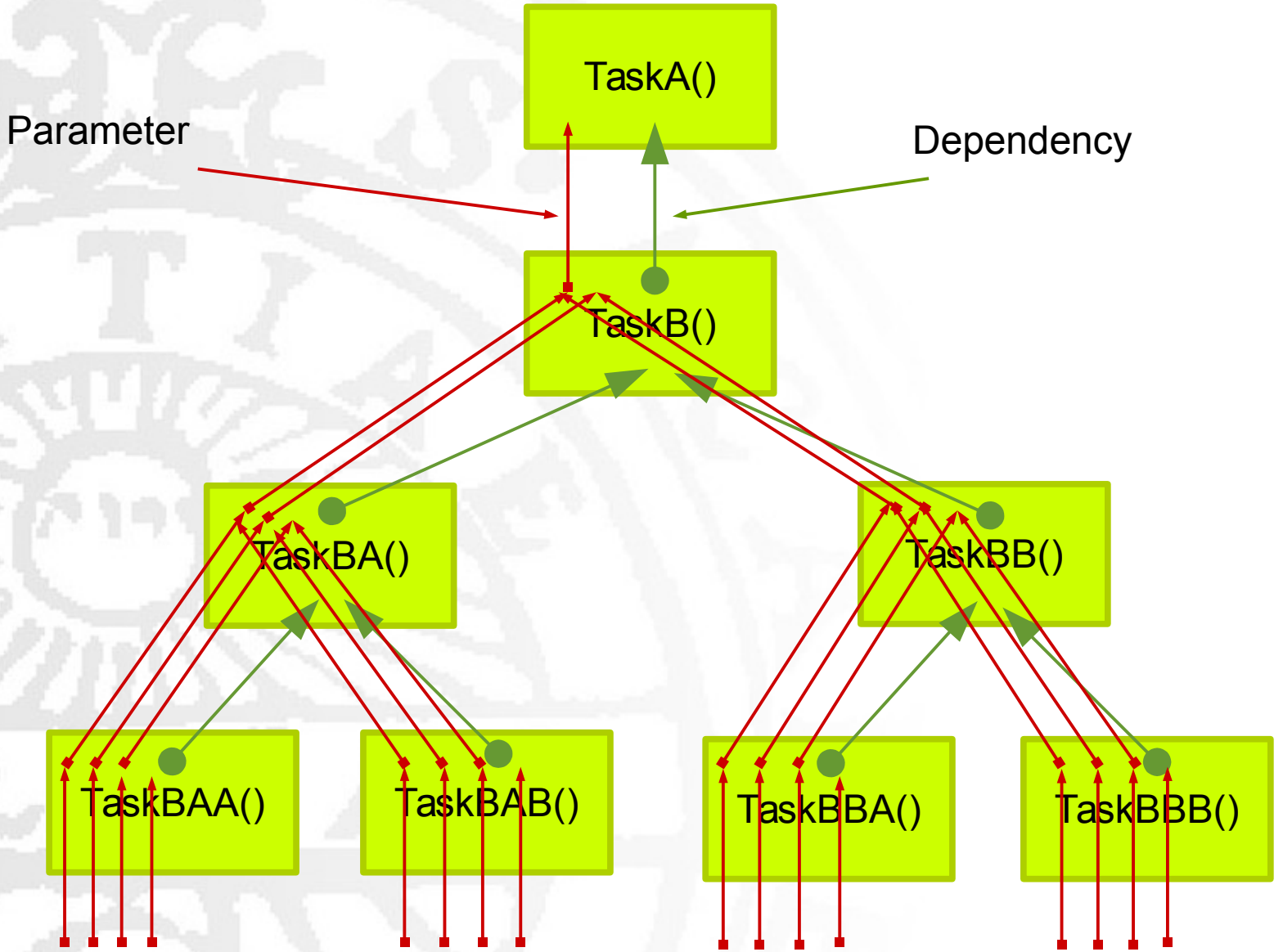
```python
import luigi

# A luigi task
class HelloWorld(luigi.Task):
    def requires(self):
        return None
    def output(self):
        return luigi.LocalTarget('helloworld.txt')
    def run(self):
        with self.output().open('w') as outfile:
            outfile.write('Hello World!\n')

# Another task, that depends on the above one
class NameSubstituter(luigi.Task):
    name = luigi.Parameter()

    def requires(self):
        return HelloWorld() # <-- Dependency definition
    def output(self):
        return luigi.LocalTarget(self.input().path + '.name_' + self.name)
    def run(self):
        with self.input().open() as infile, self.output().open('w') as outfile:
            text = infile.read()
            text = text.replace('World', self.name)
            outfile.write(text)

if __name__ == '__main__':
    luigi.run()

```

UPPSALA UNIVERSITET

# Dependencies: Default way

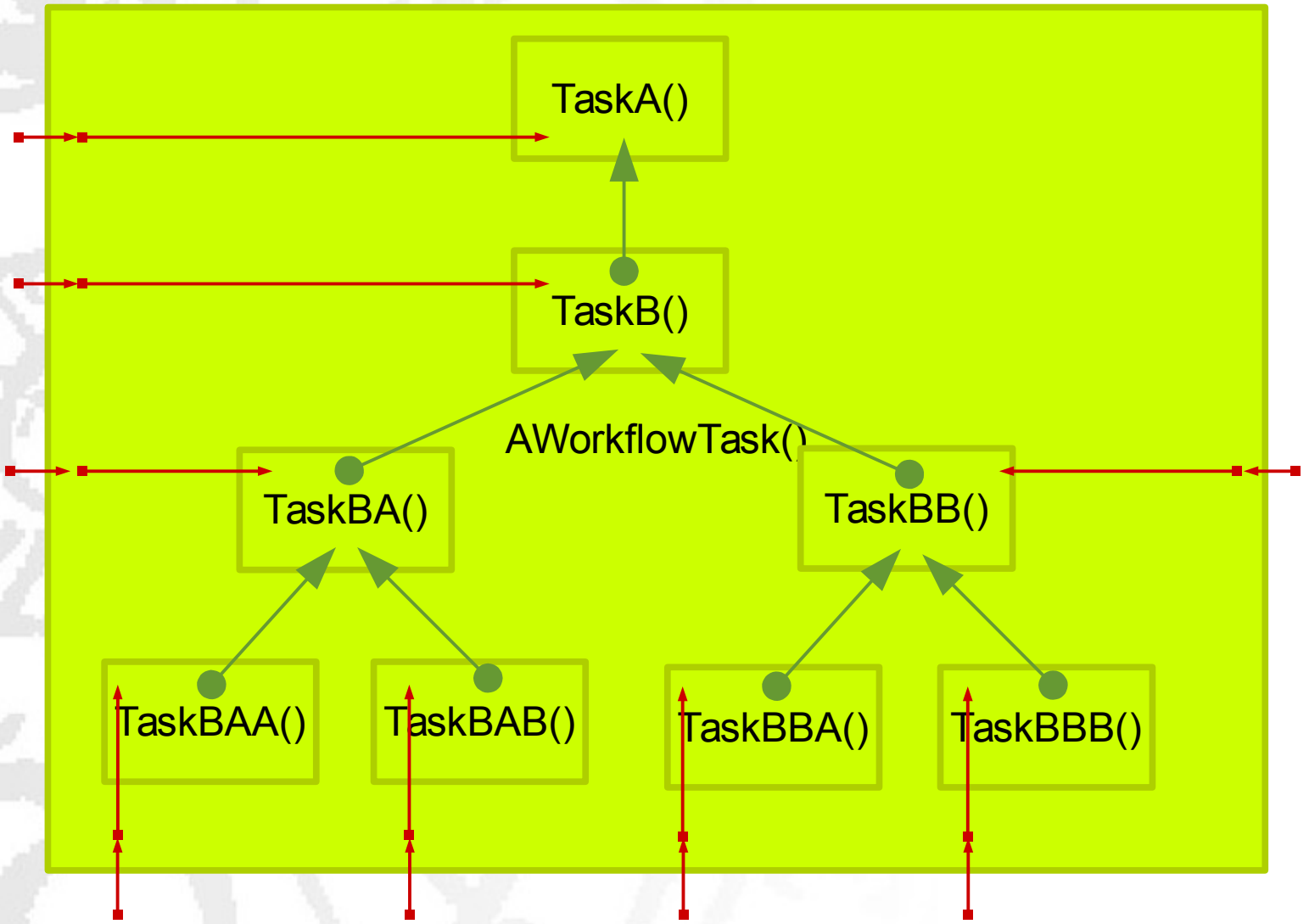


# Parameters: Default way



# Growing list of parameters

# Dependencies and parameters: **A better way**



# Supporting separate network definition

```
53  class DependencyMetaTask(luigi.Task):
54      # METHODS FOR AUTOMATING DEPENDENCY MANAGEMENT
55      def get_upstream_targets(self):
56          upstream_tasks = []
57          for param_val in self.param_args:
58              if type(param_val) is dict:
59                  if 'upstream' in param_val:
60                      upstream_tasks.append(param_val['upstream']['task'])
61          return upstream_tasks
62
63      def requires(self):
64          return self.get_upstream_targets()
65
66      def get_input(self, input_name):
67          param = self.param_kwargs[input_name]
68          if type(param) is dict and 'upstream' in param:
69              return param['upstream']['task'].output()[param['upstream']['port']]
70          else:
71              return param
72
73      def get_value(self, input_name):
74          param = self.param_kwargs[input_name]
75          if type(param) is dict and 'upstream' in param:
76              input_target = param['upstream']['task'].output()[param['upstream']['port']]
77              if os.path.isfile(input_target.path):
78                  with input_target.open() as infile:
79                      csv_reader = csv.reader(infile)
80                      for row in csv_reader:
81                          if row[0] == param['upstream']['key']:
82                              return row[1]
83              else:
84                  return 'NA'
85          else:
86              return param
87
```

UPPSALA UNIVERSITET

# Using this functionality in tasks

```
1  class CreateSparseTrainDataset(DependencyMetaTask, TaskHelpers, DatasetNameMixin, AuditTrailMixin):
2
3      # INPUT TARGETS
4      train_dataset_target = luigi.Parameter()
5
6      # TASK PARAMETERS
7      replicate_id = luigi.Parameter()
8
9      # DEFINE OUTPUTS
10     def output(self):
11         basepath = self.get_input('train_dataset_target').path
12         return { "sparse_train_dataset" : luigi.LocalTarget(basepath + ".csr"),
13                  "signatures" : luigi.LocalTarget(basepath + ".signatures"),
14                  "log" : luigi.LocalTarget(basepath + ".csr.log") }
15
16     # WHAT THE TASK DOES
17     def run(self):
18         self.x([JAVA_PATH, "-jar jars/CreateSparseDataset.jar",
19                 "-inputfile", self.get_input('train_dataset_target').path,
20                 "-datasetfile", self.output()['sparse_train_dataset'].path,
21                 "-signaturesoutfile", self.output()["signatures"].path,
22                 "-silent"])
23
```

# Using this new functionality in workflows, and wrapping whole workflows in Tasks

```
7  class MMSVMWorkflow(luigi.Task):
8      '''
9      This class runs the MM Workflow using Support Vector Machine
10     as the method for doing machine learning
11     '''
12
13     # WORKFLOW PARAMETERS
14     dataset_name = luigi.Parameter()
15     replicate_id = luigi.Parameter()
16     test_size = luigi.Parameter()
17     train_size = luigi.Parameter()
18     sampling_seed = luigi.Parameter(default=None)
19     sampling_method = luigi.Parameter()
20     svm_gamma = luigi.Parameter()
21     svm_cost = luigi.Parameter()
22     accounted_project = luigi.Parameter()
23     parallel_svm_train = luigi.BooleanParameter()
24     #folds_count = luigi.Parameter()
25     svm_type = luigi.Parameter()
26     svm_kernel_type = luigi.Parameter()
27
28     def __init__(self, *args, **kwargs):
29         super(MMSVMWorkflow, self).__init__(*args, **kwargs)
30         '''
31         The dependency graph is defined here!
32         '''
33         # ------------------------------------------------------------------------
34         self.existing_smiles = ExistingSmiles(
35                 dataset_name = self.dataset_name,
36                 replicate_id = self.replicate_id,
37                 accounted_project = self.accounted_project)
38         # ------------------------------------------------------------------------
39         self.gen_sign_filter_subst = GenerateSignaturesFilterSubstances(smiles_target=
40                 { 'upstream' : { 'task' : self.existing_smiles,
41                                  'port' : 'smiles' } },
42                 min_height = 1,
43                 max_height = 3,
44                 dataset_name = self.dataset_name,
45                 replicate_id = self.replicate_id,
46                 accounted_project = self.accounted_project)
```

# Unit testing Luigi tasks is easy

```python
import os
import time

class TestConcatenate2Files():
    file1_path = '/tmp/luigi_concat2files_file1'
    file2_path = '/tmp/luigi_concat2files_file2'

    file1_content = 'A'*80 + '\n'
    file2_content = 'B'*80 + '\n'

    def setup(self):
        with open(self.file1_path,'w') as file1:
            file1.write(self.file1_content)
        with open(self.file2_path,'w') as file2:
            file2.write(self.file2_content)

        self.concat2files = Concatenate2Files(
                replicate_id='TESTID',
                accounted_project='b2015002',
                file1_target=luigi.LocalTarget(self.file1_path),
                file2_target=luigi.LocalTarget(self.file2_path),
                skip_file1_header=False,
                skip_file2_header=False
        )

    def teardown(self):
        os.remove(self.file1_path)
        os.remove(self.file2_path)
        os.remove(self.concat2files.output()['concatenated_file'].path)

    def test_run(self):
        # Run the task with a luigi worker
        w = luigi.worker.Worker()
        w.add(self.concat2files)
        w.run()
        w.stop()

        with open(self.concat2files.output()['concatenated_file'].path) as concat_file:
            concatenated_content = concat_file.read()

        assert concatenated_content == self.file1_content + self.file2_content

```

UPPSALA UNIVERSITET

# Lessons Learned (April 2014)

- Only “pull” or “call/return” semantics, no “push” or “auto triggering by inputs”
- Not fully separate workflow
- Multiple inputs / outputs somewhat error-prone
- Dynamic typed language
- Check out for max processes limits
- No real distribution of compute or data (only common scheduling)

UPPSALA
UNIVERSITET

# Lessons Learned (Update **Jan 2015**)

- Only “pull” or “call/return” semantics, no “push” or “auto triggering by inputs”
- Not fully separate workflow def.
  - **We found a way around this.**
- Multiple inputs / outputs somewhat error-prone
  - **We're looking into solving this too.**
- Dynamic typed language
  - **We've found that unit testing is easy.**
- Check out for max processes limits
- No real distribution of compute or data (only common scheduling)
  - **We have found ourselves running everything through SLURM anyway**

UPPSALA UNIVERSITET

Thank you!

UPPSALA
UNIVERSITET