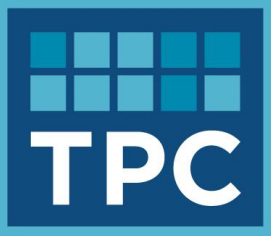


TAX POLICY CENTER
URBAN INSTITUTE & BROOKINGS INSTITUTION

SCALING MICROSIMULATION MODELING IN THE CLOUD: A GUIDE

Jessica A. Kelly, Kyle Ueyama, and Alyssa Harris

August 30, 2019

ABSTRACT

This document introduces the goals of Modeling in the Cloud at the Urban Institute. It details how our cloud-based system uses continuous deployment, containers, an interface, and computing and storage components that leverage Amazon Web Services (AWS), then lists lessons learned and critical decision points. It also provides code for a simple model, several components for public use, and instructions for installing and running the simple model.

ABOUT THE TAX POLICY CENTER

The Urban-Brookings Tax Policy Center aims to provide independent analyses of current and longer-term tax issues and to communicate its analyses to the public and to policymakers in a timely and accessible manner. The Center combines top national experts in tax, expenditure, budget policy, and microsimulation modeling to concentrate on areas of tax policy that are critical to future debate.

CONTENTS

ACKNOWLEDGMENTS

The Tax Policy Center thanks the Alfred P. Sloan Foundation for its support of this work.

The views expressed are those of the authors and should not be attributed to the Urban-Brookings Tax Policy Center, the Urban Institute, the Brookings Institution, their trustees, or their funders. Funders do not determine research findings or the insights and recommendations of our experts. Further information on Urban's funding principles is available at http://www.urban.org/aboutus/our-funding/funding-principles; further information on Brookings's donor guidelines is available at http://www.brookings.edu/support-brookings/donor-guidelines.

The authors would like to thank Len Burman, Khuloud Odeh, Ramani Gadde, Robert McClelland, Daniel Berger, Philip Stallworth, and Chenxi Lu; and our partners at Vizuri and Levvel for their contributions.

INTRODUCTION

The Urban Institute has developed several microsimulation models for different policy centers and clients. These models have been built in different languages and have varying inputs and outputs. We are beginning to standardize our model development and deployment protocols and to build an architecture that allows us to run simulations both faster and in greater volume. Our long-term goal is to build an architecture that can accommodate all models, regardless of programming language, inputs, or outputs.

While each model at Urban has a unique and often long-standing and homegrown ecosystem, there are shared cross-model requirements and activities. One model might produce highly detailed, formatted output using specialized Fortran, while another might render data using a web interface that connects on a database—yet they both produce summary tables. One might use a very specifically formatted and complicated CSV file to specify runtime options, while another might use a database to hold program rules—yet they both have execution options and rules. These commonalities allow for the development of a standard architectural pattern across models.

But why make changes to existing and functioning models and frameworks? Consider the power of not just running one simulation to estimate marginal tax rates on labor income given a small change in the earned income tax credit, but of running it 100 times to find the most desirable possible changes to the Earned Income Tax Credit. Assume each run takes 10 minutes on a personal computer. Done sequentially, these runs would take 100 × 10 = 1,000 minutes (16 hours and 40 minutes). If we could not only speed up a single run—say, cutting it in half to 5 minutes—but also run 10 models simultaneously in parallel, this would cut runtime to 5 × 10 = 50 minutes. If we could run all 100 in parallel, this would cut runtime to 5 minutes! The best way to do this cost-effectively is to use the cloud.

Goals we kept in mind while developing this Modeling in the Cloud system are as follows:

- Minimize the disruption to current model development. The first model we lifted to the cloud was one that has been developed over many years and had a well-established code base. Our choices were made such that most of this code could remain in its current state, so the modeling staff could continue to work in accustomed fashion with only minor workflow changes (e.g., we introduced a branching model in GitHub).
- Have capacity to handle bigger datasets (scale vertically).
- Have capacity to handle large amounts of runs (scale horizontally).
- Be cost-effective.
- Allow focus of expertise (technical staff focuses on technical, analysts focus on analyzing).

This project focuses on what it would take to deploy an Urban Institute model to the cloud, and in particular, what is required to scale horizontally for more runs. Using the architecture outlined here, we have been able to reduce runtime by a minimum factor of 65 (likely higher, but hard to estimate), and can run jobs for under $0.02 per run.

In this report, we first explore what it takes to move a model that has traditionally run on a local computer to the cloud, what technology components are used to manage cloud runs, and the steps needed to enact thousands of runs as a group. Next, we discuss the limitations of the current system and the costs to run it, along with some critical lessons learned from the process. Last, we present a Simple Tax Model to provide code for public use and to illustrate how models are run from the cloud.

CLOUD INFRASTRUCTURE BASICS AND TERMINOLOGY

Using the cloud makes it easier to build flexible infrastructure while managing costs. In the past, one might have purchased one or more powerful servers to process many runs. Yet this machinery might also sit idle for extended periods when this level of power is not needed, degrading over time and becoming obsolete. By instead using cloud services, we can spin up more computing power only when needed, and easily change the machine type as upgrades are made.

In general, Urban uses Amazon Web Services (AWS), but we also use other cloud providers like Azure or Google, all of which offer tools that perform similar functions.

This document will refer both to specific technology components used to create our cloud-based modeling infrastructure (e.g., AWS EC2 instances) and to technical terms about concepts (e.g., containers). Definitions and further resources are listed next, and links to official documentation are provided in the "Resources" section at the end of this document.

AWS-specific terminology

- AWS: Amazon Web Services, a cloud provider
- EC2: Elastic Cloud Compute. This AWS service enables users to create and manage instances of virtual servers in AWS.
- EBS: Elastic Block Store. This AWS service provides storage volumes used in conjunction with EC2 instances. This storage is faster for instances to access than S3 (see below).
- RDS: Relational Database Service. This AWS service makes creating and maintaining a relational database (e.g., MySQL or Oracle) easier to set up in the cloud.
- S3: Simple Storage Service. This AWS service provides an easy storage system for any type of file and makes it accessible via a URL.
- Lambda: This AWS service allows users to create serverless functions that can be run on demand or via triggers.

Other technical terms

- JSON: JavaScript Object Notation. A common way of defining data. Each element consists of a key and a value.
- Git and GitHub: Git is an open-source version control system, and GitHub is a service and website that provides access to Git repositories and adds workflow and other services.

- Instance: In relation to AWS, this is one virtual server and can be thought of in the same way as one would think of a single server.
- Container: A technology that enables the definition of an entire system, from the operating system to components and programs to run upon creation.
- Docker and DockerHub: Docker is a program that enables the creation and management of containers. Much like GitHub, DockerHub facilitates the sharing of Docker images. It also provides tools for automation and connections to GitHub.
- Rancher: An open-source platform for managing containers.

We also provide code snippets, which are formatted this way:

```
echo "Hello World"
```

VERSIONING AND ENCAPSULATING THE MODEL

The first step to running the model on the cloud is, of course, enabling the code to run seamlessly on a cloud server. This process has three steps: using a version control system that will ensure the code is the same across all developers and users, programmatically generating a runtime environment for the model, and establishing a database of all the successfully built versions of the model that are available to be run.

STEP 1: USING VERSION CONTROL WITH GITHUB AND CREATING A DOCKERFILE TO PROGRAMMATICALLY GENERATE A RUNTIME ENVIRONMENT. We are able to achieve this standard runtime environment using containers, which permit models to be run anywhere—on a users' local computer, a virtual server, or a cloud server. One standard way to define a container is to use Docker. Our GitHub repository contains both the model code and the Dockerfile to define the environment and load the code.

For example, a Dockerfile that defines a Fortran-based model called ExampleModel follows:

```
FROM centos:7
RUN yum update -y && yum install -y gcc-gfortran gdb make && yum clean all

WORKDIR /ExampleModel/
COPY * /ExampleModel/
COPY /Fortran /ExampleModel/Fortran/

RUN make

ENTRYPOINT [ "/ExampleModel/Run_job.sh" ]
```

This code first creates a container image based on the Centos operating system, then adds Fortran components, copies Fortran files from the local environment into the image, runs make to compile the code, and then defines the shell script that will then execute the command to run the model. The image only creates a template of what is needed to run the model; it does not run the model upon its creation, running the model occurs only when the container image is run. Multiple containers can be created and run from the same image. This is a fine distinction but should be noted.

STEP 2: AUTOMATING THE DOCKER BUILD WITH DOCKERHUB. We can further automate the process by storing this container image in DockerHub and connecting it to the GitHub repository. This ensures that the template will be created each time a developer pushes either code with a new model version, or edits to an existing model, to the GitHub repository. One way of doing this is creating an "autobuild" triggered by a "webhook."[1] A detailed example of this implementation is provided in the "Example Cloud-Based Model: The Simple Tax Model" section. With this set up, once code is pushed to a GitHub repository, it triggers a build of a Docker image on DockerHub.

STEP 3: ADDING THE MODEL VERSION TO A DATABASE. Once the image build is complete, a second webhook executes and enters the image name into a database, which gives us a list of models available to run. The way this process is designed will vary by use case. Urban has many different production versions of the Tax Policy Center (TPC) model that need to be available for use at any time. In other cases, there might be only one version of a model, which would greatly simplify this step.

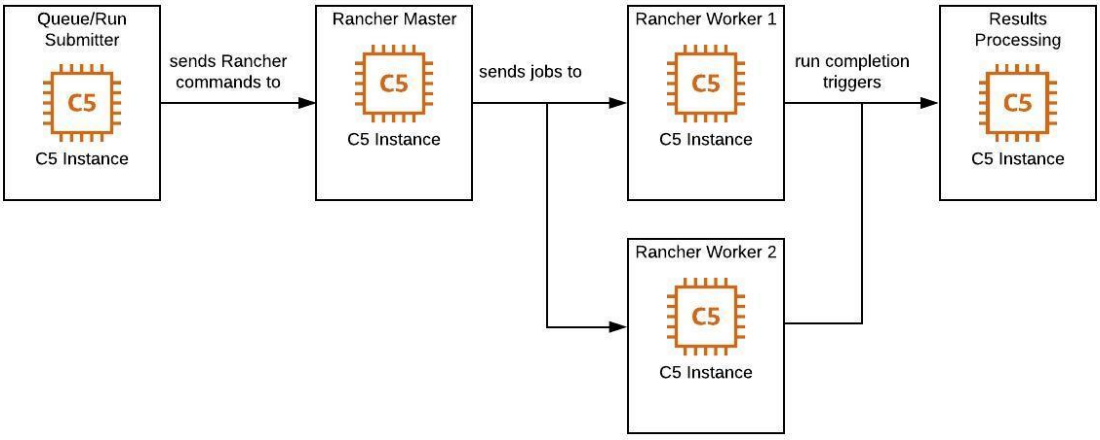
USING RANCHER AS ORCHESTRATOR AND CONTAINER SETUP

We use Rancher to submit individual runs of the TPC model. It is important to note that we use Rancher version 1.6, and also use Rancher's Cattle tool to manage, or "orchestrate," our runs. Rancher 2.0 represents a major overhaul and is focused toward Kubernetes, which is now the only orchestration framework, so be careful when reading additional documentation to look at the correct version.[2]

At present, we use one master node to gather and send runs to individual worker nodes. Each of these worker nodes runs a set of jobs simultaneously, and as jobs finish, processes additional jobs as sent by the master node. We use a separate AWS EC2 queue-submitter instance to manage sending Rancher commands to the master node, since all the rancher-compose files that define the jobs are created by code; similarly, we use a separate AWS EC2 instance to process results as jobs are finished (see figure 1).

FIGURE 1

AWS EC2 Instances used to Manage Rancher and Results



The TPC model itself is not a single container, but actually a set of six containers that manage inputs, move specific files from one AWS location in S3 into the main container, wait for the model to be completed, and copy files out of the container to another S3 location. These are all controlled and defined inside the Rancher and Docker Compose YAML definition files. For illustration purposes, we will use a single container with the Simple Tax Models in the "Example Cloud-Based Model: The Simple Tax Model" section. What follows is one such YAML definition file, `docker-compose-job-submitter-template.yml`, that shows all the containers together in the definition file:

```
version: '2'
services:
   modelrun:
      image: ourockerhubaccount/tpcmodelmaster:{TAG}
      network_mode: "none"
      container_name: {MIC_JOB_ID}
      volumes_from:
         - input
         - output
         - param
         - locking
# important to set constraints to allow more than a single run at a time per host.
      cpu_shares: 1024
      mem_limit: 2147483648
      memswap_limit: 4294967296
      labels:
         io.rancher.scheduler.affinity:host_label: MIC-Use=TPC_Modeling,MIC-
Worker={MIC_WORKER}
         io.rancher.container.pull_image: always
         io.rancher.container.start_once: true
         io.rancher.sidekicks: locking, input, output, param, input-helper
      command: [ "{MIC_JOB_ID}", "30000", "{PARAMETER_FILE}","{START_YEAR}", "{END_YEAR}" ]

# internal ordering of operations
   locking:
      image: busybox
      network_mode: "none"
      volumes:
         - /locks
      labels:
         io.rancher.container.start_once: true

# I/O containers
   input:
      image: ourockerhubaccount/uiio_input
      network_mode: "bridge"
      volumes_from:
         - locking
      labels:
         io.rancher.container.start_once: true
         #io.rancher.container.pull_image: always

   param:
      image: ourockerhubaccount/uiio_param
      network_mode: "bridge"
```

```
      command: [ "s3://ourbucket/tpc/param/{PARAMETER_FILE}" ]
      volumes_from:
         - locking
      labels:
         io.rancher.container.start_once: true
         io.rancher.container.pull_image: always

   output:
      image: ourockerhubaccount/uiio_output
      network_mode: "bridge"
      command: [ "s3://ourbucket/tpc/results/{MIC_JOB_ID}/" ]
      volumes_from:
         - locking
      labels:
         io.rancher.container.start_once: true
         #io.rancher.container.pull_image: always

# Gathers the input file details from the parameter list
   input-helper:
      image: ourockerhubaccount/uiio_input-helper
      network_mode: "none"
      command: [ "{PARAMETER_FILE}" ]
      volumes_from:
         - locking
         - param
      labels:
         io.rancher.container.start_once: true
```

The container functions are as follows:

- `modelrun`: This container actually runs the Fortran model.
- `locking`: Controls program flow through a series of file locks. This ensures that all containers run in order, because by default they will all want to start at the same time. But first we need to gather inputs and parameters, then run the model, then gather and save outputs.
- `input`: Copies all the input files needed for a model run. These are loaded and shared under the `/data` directory. This flow is controlled by the locking container.
- `param`: The parameter files are loaded and shared under the `/param` directory inside of the container. Files are copied to this location from S3 and are available before the `modelrun` container starts. This flow is controlled by the locking container.
- `output`: All output files are expected to be written into the `/results` directory inside of the container. The files in this location are staged up to S3 upon job completion. This flow is controlled by the locking container.
- `input-helper`: Gets information about filenames from the parameter file, and was once used to assist the input container.[3] As with the other containers, flow is controlled by the locking container.

There are some important details in these container definitions that have been specified as a result of issues uncovered as we have used the system. We note these in the "Lessons Learned" section.

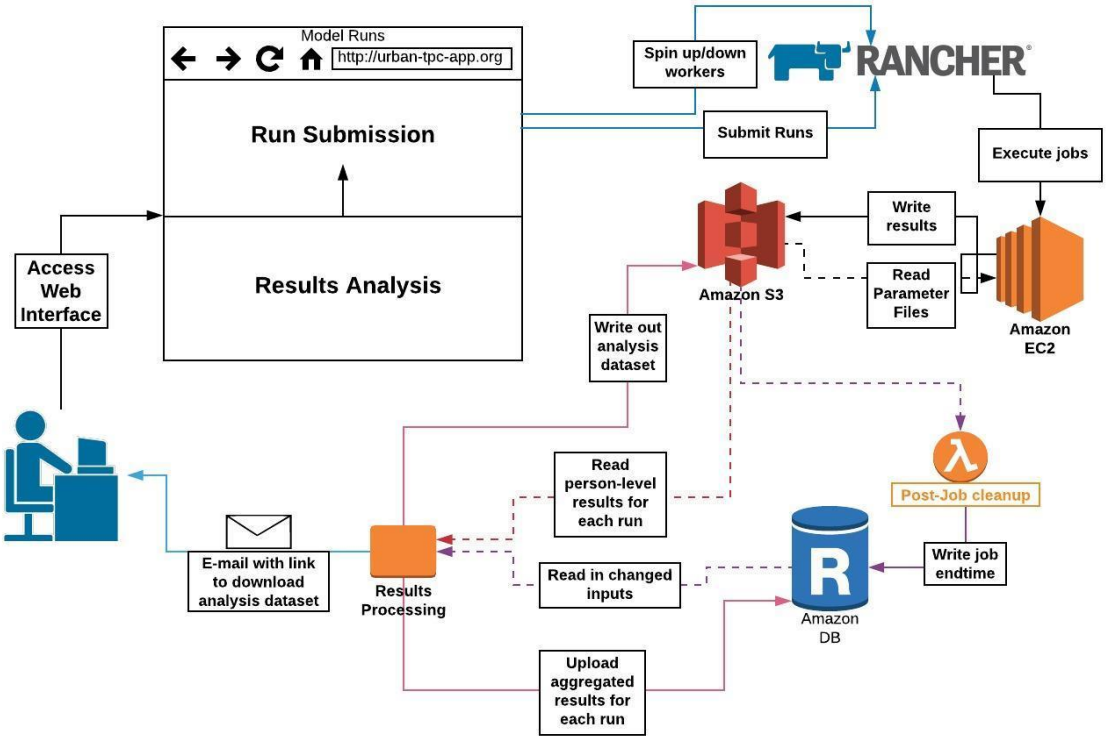
Our job submission has been set up such that `rancher-compose` files are created programmatically to send job stacks to the Rancher infrastructure. This is simply taking the `docker-compose-job-submitter-template.yml` file as a starting point and specifying values for tagging the AWS resources, and the values for `MIC_JOB_ID`, `MIC_WORKER`, `PARAMETER_FILE`, `START_YEAR`, and `END_YEAR`. This is detailed in the next section.

PUTTING IT ALL TOGETHER, RUNNING THOUSANDS OF TIMES

With these initial components complete, and because we can use code to define our runtime environment, we can begin to expand from running one model to programmatically running thousands. The full system is shown in figure 2. This workflow shows what happens after the model has been pushed to GitHub, the Docker build triggers, and the image is registered to the database. Most of the actions are triggered by AWS Lambda functions. These are small functions that run without a server attached and ensure that our process runs in proper order.

FIGURE 2

Model Submission Components and Workflow



Generate Parameters: A Presubmission Step

Because the TPC model has many hundreds of parameters stored in a CSV file, it would be prohibitive and inefficient to edit and create hundreds or thousands of variations by hand. To solve this problem, parameter files are created using code. This step alone, while seemingly simple, was one of the key components that allowed for thousands of model runs to be run so easily. The following code leverages a module created specifically for the TPC model parameter file, and is an example of what an analyst would run to create a set of parameter files.

```
from ParamUpdater import ParamUpdater # load module

# create parameter object from an existing file
params = ParamUpdater('original/cli-poc.csv', credentials)

# do updates
# increment standard deduction by 100
for i in range(params.get_value('STANDARD',0)+100,2000,100):
    params.update('STANDARD',i)
    filename = 'parameterOptions_Standard_' + str(i)
    params.write_modified(filename)
```

This code not only creates the parameter files with the correct naming convention, but also uploads them to the correct location in S3 to be used later when defining a run. The module tracks which parameters have been changed compared to the file defined as the original file, and writes these changes to the Inputs table of the RDS database. This is what the results processing will query when it creates the full analysis dataframe.

We also provide a small example of this concept with a simplified underlying Python module, `ParamUpdater`, in the "Example Cloud-Based Model: The Simple Tax Model" section.

Access the Web Interface

Users define their run as a collection of jobs via a simple RShiny interface. We chose this route because it was easy for us to prototype and develop using this framework. However, this interface could be written and created in any web language. In the interface, the user enters the following:

- email address (for notifications)
- run name and description
- the branch/version of the model to use
- a parameter file stub (prefix used in the `ParamUpdater` step just described)
- start and end year
- number of worker nodes to add

As the user enters this information, the interface uses the parameter file stub specified—the prefix used by the code that programmatically generates parameter files—to pull a list of those parameter files from an S3

location, `tpc/param`, that begin with the given stub. We can count on these files being in this location because we have written them there in the `ParamUpdater` step. The interface further ensures that the length of that list is 2,000 jobs or less.[4] Additionally, using the number of parameter files to be run, the interface makes a recommendation on the number of worker instances to add. This is based on our experience with our particular model.

When the user submits the form, details are written to a JSON file in the S3 location `tpc/queue` to be picked up in the next step.

Queue Creation and Spinning Up/Down Workers

When the interface submission JSON file is written to S3 in `tpc/queue`, it triggers an AWS Lambda function in Python called `tpc-job-queue.py`. This function reads the submission file, creates the specified number of workers, and adds them as worker nodes to the Rancher interface. The `boto3` library handles the AWS specifications for the instance. The following is a snippet of this code; portions have been removed for security and brevity and are noted with ellipses. The other AWS Lambda functions follow this same design pattern, with slight variations on how exactly the hosts are created.

```
import boto3
import json
import os

# get correct bootstrap script
def get_bootstrap(os, bucket):
    s3 = boto3.client('s3', region_name = 'us-east-1')
    os_key = 'tpc/os/{}.sh'.format(os)
    bootstrap_script = s3.get_object(Bucket = bucket, Key = os_key)
    return bootstrap_script['Body'].read()

# main lambda function
def lambda_handler(event, context):
    # set rancher url
    RANCHER_URL = 'https://our/url/v2-beta/projects/1a7/hosts'
    RANCHER_ACCESS_KEY = os.environ.get("RANCHER_ACCESS_KEY")
    RANCHER_SECRET_KEY = os.environ.get("RANCHER_SECRET_KEY")

    # create boto3 drivers
    ec2 = boto3.resource('ec2', region_name = 'us-east-1')
    s3 = boto3.client('s3', region_name = 'us-east-1')

    # get s3 bucket and object key from trigger
    BUCKET = event['Records'][0]['s3']['bucket']['name']
    KEY = event['Records'][0]['s3']['object']['key']
    RUN = KEY.split('/')[-1]

    # get run submission
    obj = s3.get_object(Bucket = BUCKET, Key = KEY)
    submission = json.load(obj['Body'])

    # get number of hosts to spinup
    HOSTS = submission['workers']
```

```
    TAG = submission['body']['tag']

    # set instance defaults
    ...
    # set ebs storage
    BLOCK_DEVICE_MAPPINGS = [
        {
            'DeviceName' : '/dev/xvda',
            'Ebs': {
                'VolumeSize': STORAGE,
                'VolumeType': 'gp2'
            }
        }
    ]

    # set IAM role
    IAM = {
        'Name': IAM_ROLE
    }

    # set network interfaces
    NETWORK_INTERFACES=[
        {
            "DeviceIndex": 0,
            "SubnetId": SUBNET_ID,
            'Groups': SECURITY_GROUPS,
            "AssociatePublicIpAddress": True
    }]

    # get current number of workers
    r = requests.get(RANCHER_URL, auth = (RANCHER_ACCESS_KEY, RANCHER_SECRET_KEY))
    num_host = len(r.json()['data'])

    ...

    # create host
    for i in range(0, HOSTS):
        # update bootstrap script with proper worker number
        ...
        # set instance tags
        ...
    # increment num_host
        num_host += 1
    # create instance
        instances = ec2.create_instances(ImageId = IMAGE_ID,
                                 InstanceType = INSTANCE_TYPE,
                                 TagSpecifications = TAG_SPECIFICATIONS,
                                 KeyName = KEY_NAME,
                                 IamInstanceProfile = IAM,
                                 MinCount = 1,
                                 MaxCount = 1,
                                 UserData = USER_DATA,
                                 NetworkInterfaces = NETWORK_INTERFACES,
                                 BlockDeviceMappings = BLOCK_DEVICE_MAPPINGS)
```

A startup "bootstrap" script (get_bootstrap in the preceding code), run on the creation of the instance, handles the Docker and Rancher setup in order to add the new worker nodes to the Rancher infrastructure, and pulls the Docker images required. This ensures a faster initialization time when the model runs begin.

When the final instance has been fully spun up, the submission JSON file is copied over to another S3 location, tpc/submissions, to trigger the next step, as we want to be sure the code doesn't start submitting runs until the worker nodes are created and ready.

Submit Runs

When the interface submission file is copied to the S3 location `tpc/submissions`, it triggers an AWS Lambda function in Python called `tpc-run-submit.py`. This function reads the submission file, creates an instance to submit the model run jobs to Rancher, creates another instance to process the results, and prepares for the model runs to start. As with the queue step, the `boto3` in Python library handles the AWS specifications for the instance.

A startup bootstrap script on creation of the instance installs Python and rancher-compose, copies the necessary files into the instance, and starts the model run job-submission loop that waits for runs to process.

Additionally, for the instance that processes results, a bootstrap script installs Python, copies the necessary files into the instance, and starts the code loop that listens for results to process and then processes them.

Execute Jobs

As noted earlier in "Using Rancher as Orchestrator and Container Setup," the execution of a model run takes a `docker-compose` YAML template, reads the submission JSON file, and iteratively calls `rancher-compose` to send job stacks to the Rancher infrastructure. These jobs are executed on the worker nodes and triggered by the model run job-start in the "Submit Runs" step.

As jobs are submitted, their details (except end time) are also written to a table in the RDS database. Each job will write its output files to an S3 location called `tpc/results`. Once all files have completed writing, an AWS Lambda function `tpc-stack-remove.py` is triggered. This function will write the model end time to the RDS database and remove the stack from Rancher. This last step is critical: if this is not done, the worker instance will run out of space and crash.

Write and Process Results

Running simultaneously with the worker nodes running model jobs, the analysis instance runs a Python script that queries the Runs table in the RDS database. To define the output file, a small CSV file, `extract.csv`, is used to specify which micro-level variables are output. Unlike the parameter file, this file is the same across all model jobs. As job stacks complete, the `extract.csv` is read in and summary results are calculated using Python. A snippet of this code is shown below (note that we can and do leverage multiprocessing within Python in this step).[5] Cut portions are marked with ellipses.

```
# -*- coding: utf-8 -*-
"""
Created on Fri Feb 23 16:16:42 2018

@author: aharris

This script interacts with the following database tables in:
    <Database table> : <Columns>
    Inputs: ParamFile, Param, ParamValue, Year
    Results: JobID, ResultVar, ResultValue
The resulting dataframe is saved to a CSV on S3

"""
import pandas as pd
import pymysql
import time
import os
import sys
import io
import boto3
import botocore
import logging
from status_enum import RunStatus
import tpc_upload_results
import multiprocessing as mp
import csv

from ec2credentials import  usern, passwordv

from status_email import send_email

logging.basicConfig(filename='Results.log',level=logging.DEBUG)
logging.debug('Begin processing results')

run_name = sys.argv[1]
update = sys.argv[2]
if update == "True":
    update = True
else:
    update = False

s3 = boto3.client('s3', region_name = 'us-east-1')
logging.debug("Established s3 connection")

logging.debug("Established DB connection")
start = time.time()

keepGoing= True
keepCheckingSQL= True

job_dfs = []
counter = 0

complete_sql = ""
failed_sql = ""

email_since = None
```

```
email_period = 3600

email_lock = mp.Lock()
writing_lock = mp.Lock()

...

def processJobs(job, counter, email_since, user_email, status, tot_jobs, first_email,
completed_jobs, failed_jobs):
    print("Processing job {}: {}".format(counter, job))
    try:
        # Download the extract and process
        s3.download_file('ourbucket', 'tpc/results/' + str(job) + '/ExtractFile.csv',
'jobs/' + str(job) + '.csv')
        conn = dbConnection()
        # Run main functions from tpc-process-results
        tpc_upload_results.resultmain_loading(job, conn)
        with email_lock:
            if time.time() - first_email.value >= email_period:
                first_email.value = time.time()
                email_since = check_status(run_name, user_email, email_since, status,
tot_jobs, completed_jobs, update, failed_jobs)
        tpc_upload_results.resultmain_setup(job, conn)
        with email_lock:
            if time.time() - first_email.value >= email_period:
                first_email.value = time.time()
                email_since = check_status(run_name, user_email, email_since, status,
tot_jobs, completed_jobs, update, failed_jobs)
        tpc_upload_results.resultmain_make(job, conn)
        with email_lock:
            if time.time() - first_email.value >= email_period:
                first_email.value = time.time()
                email_since = check_status(run_name, user_email, email_since, status,
tot_jobs, completed_jobs, update, failed_jobs)
        os.remove('jobs/' + job + '.csv')
        # Link the inputs and results
        create_df(job, conn)
        conn.close()
        completed_jobs.append(job)
    except botocore.exceptions.ClientError as e:
        failed_jobs.append(job)
        print("There was a ClientError on job {}".format(job))
        logging.warning("There was a ClientError on job {}".format(job))
    except IOError as e:
        failed_jobs.append(job)
        logging.error("I/O error({}): {}".format(e.errno, e.strerror))
    except:
        failed_jobs.append(job)
        logging.error("Unexpected error:" + str(sys.exc_info()[0]))
        logging.error("Error processing job " + str(job))

if __name__ == '__main__':
    try:
        man = mp.Manager()
        first_email = man.Value('d', 0.0)
        completed_jobs = man.list()
        failed_jobs = man.list()
```

```
        start = time.time()
        try:
            os.remove('analysis_df.csv')
        except FileNotFoundError:
            pass
        print(run_name)
        status = RunStatus.STARTED
        print("In try")
        conn = dbConnection()
        run_info = pd.read_sql("select Email, TotalJobs from Runs WHERE `RunName` = '" +
str(run_name) + "' LIMIT 1", con=conn)
        time_begin = time.time()
        time_out = False
        while run_info.empty and not time_out:
            if time.time() - time_begin > 900:
                time_out = True
                keepGoing = False
                status = RunStatus.FAILED
                user_email = 'email@email.com'
                email_since = time_begin
                tot_jobs = 0
            else:
                # refresh the connection
                conn.close()
                conn = dbConnection()
                run_info = pd.read_sql("select Email, TotalJobs from Runs WHERE `RunName` =
'" + str(run_name) + "' LIMIT 1", con=conn)
        if not run_info.empty:
            tot_jobs = run_info['TotalJobs'][0]
            user_email = run_info['Email'][0]
            email_since = check_status(run_name, user_email, email_since, status, tot_jobs,
completed_jobs, update, failed_jobs)
            first_email.value = time.time()
            jobquery = "select JobID from Runs WHERE `RunName` = '" + str(run_name) + "'
and EndTime is not NULL"
            jobs = pd.read_sql(jobquery, con=conn)['JobID']
            conn.close()
            since = time.time()
            print("Jobs are " + str(jobs))
            while (keepGoing):
                keepCheckingSQL = True
                print("In while")
                if not jobs.empty:
                    print("Jobs aren't empty")
                    processes = []
                    status = RunStatus.IN_PROGRESS
                    for job in jobs:
                        counter +=1
                        p = mp.Process(target=processJobs, args = (job, counter,
email_since, user_email, status,

tot_jobs, first_email, completed_jobs,

failed_jobs,))
                        processes.append(p)
                        p.start()
                    for process in processes:
```

```
                        process.join()
                    since = time.time()
                    if completed_jobs:
                        complete_sql = add_job(completed_jobs)
                    if failed_jobs:
                        failed_sql = add_failed_job(failed_jobs)
                while(keepCheckingSQL):
                    jobquery = "select JobID, EndTime from Runs WHERE `RunName`= '" +
str(run_name) + "'"
                    if complete_sql:
                        jobquery += " and JobID NOT IN " + complete_sql
                    if failed_sql:
                        jobquery += " and JobID NOT IN " + failed_sql
                    conn = dbConnection()
                    jobs_sql = pd.read_sql(jobquery, con=conn)
                    conn.close()
                    print("Jobs are " + str(jobs_sql))
                    if jobs_sql.empty and (len(completed_jobs) == int(tot_jobs)):
                        print("Run has been successfully processed")
                        logging.debug("Run has been successfully processed")
                        status = RunStatus.SUCCESS
                        keepCheckingSQL = False
                        keepGoing = False
                    elif jobs_sql.empty and (len(completed_jobs) + len(failed_jobs)) ==
int(tot_jobs):
                        print("All but {} jobs were successfully
processed".format(len(failed_jobs)))
                        logging.debug("All but {} jobs were successfully
processed".format(len(failed_jobs)))
                        status = RunStatus.INCOMPLETE
                        keepCheckingSQL = False
                        keepGoing = False
                    else:
                        if not jobs_sql.empty:
                            print("Jobs_sql not empty")
                            jobs = jobs_sql[jobs_sql['EndTime'].notnull()]['JobID']
                        if jobs_sql.empty or jobs.empty: # Either jobs haven't been
initiated yet, or they haven't completed
                            print("Not yet complete")
                            print("{} jobs completed out of {}".format(len(completed_jobs),
tot_jobs))
                            elapsed_time = time.time() - since
                            # If it's been over 5 minutes since a job completed
                            print("Elapsed time is {}".format(elapsed_time))
                            if elapsed_time > 300:
                                # exit the loops
                                print("Run failed")
                                keepCheckingSQL = False
                                keepGoing = False
                                if completed_jobs:
                                    logging.warning("The following jobs were not completed:
")
                                    failed_jobs += jobs_sql['JobID']
                                    logging.warning(str(failed_jobs))
                                    status = RunStatus.INCOMPLETE
                                else:
                                    status = RunStatus.FAILED
```

```
                            else:
                                email_since = check_status(run_name, user_email,
email_since, status, tot_jobs, completed_jobs, update, failed_jobs)
                        else: # There are jobs which can be processed
                            email_since = check_status(run_name, user_email, email_since,
status, tot_jobs, completed_jobs, update, failed_jobs)
                            keepCheckingSQL = False
    finally:
        user_email = run_info['Email'][0]
        s3resource = boto3.resource('s3')
        failed_paramfiles = []
        if conn.open:
            conn.close()
        print(str(completed_jobs))
        if len(completed_jobs) > 0:
            if len(completed_jobs) != tot_jobs:
                status = RunStatus.INCOMPLETE
                if failed_jobs:
                    print("getting param files")
                    paramfilequery = "select ParameterFile from Runs where JobID in " +
"('" + "', '".join(failed_jobs) + "')"
                    conn = dbConnection()
                    failed_paramfiles = pd.read_sql(paramfilequery,
con=conn)['ParameterFile'].tolist()
                    conn.close()
            csv_buffer = io.StringIO()
            pd.read_csv('analysis_df.csv').to_csv(csv_buffer, index = False)
            s3resource.Object('ourbucket', our
resource/{}.csv'.format(run_name)).put(Body=csv_buffer.getvalue())
            logging.debug("Processing took " + str(time.time() - start) + "seconds")
        else:
            status = RunStatus.FAILED
        if status == RunStatus.INCOMPLETE or status == RunStatus.FAILED:
            logging.shutdown()
            logging.getLogger(None).handlers = []
            s3resource.Object('ourbucket',
'tpc/logs/{}.log'.format(run_name)).put(Body=open('Results.log', 'rb'))
        check_status(run_name, user_email, email_since, status, tot_jobs, completed_jobs,
update, failed_paramfiles)
        print("Overall time: {}".format(time.time() - start))
```

In this Python code, the Inputs table in RDS is queried to match the summary results with the inputs that are unique to that job's parameter file. These are appended to a CSV file. The details of results processing will depend on the particular use case, and on what sorts of outputs need to be saved and analyzed once all the model runs have completed.

While the run is processing, the user receives an email update every hour with the current progress of their run. When the run is complete, the user receives an email with a link to the results dataframe that is stored in S3. If something went wrong, the user is additionally notified in the email with a list of the parameter files that failed.

Once all results have been processed, the script calls an AWS Lambda function that terminates the EC2 instances.

BENEFITS OF THE CURRENT SYSTEM

As mentioned, manually creating parameter files to generate even hundreds of files would be tedious, error prone, and prohibitively long in duration. Along with saving time, this code development makes it easier to see what parameter changes have been made, permits runs to be associated with specific changes (vital for analysis, as it reliably connects a result to a particular parameter change), and permits anyone to review what was done for any given changes. The code can also be applied to day-to-day model development.

We have increased the number of daily runs possible by a minimum factor of 65, assuming that we could generate 2,500 parameter files in 5 minutes per file, and that we could run four models simultaneously in about 5 minutes. This would take 15,625 minutes to do, or approximately 260.4 hours. (2,500 × 5 = 12,500 minutes for parameter files and 625 × 5 = 3,125 minutes for the runs). This all assumes that we made no mistakes in the parameter files, or in formulating the commands for executing the models. Still to be worked out are how to organize the model run output and put together an analysis file. Using the cloud architecture, we can produce an analysis file that links results back to the specific parameters that were changed over 2,500 runs in about 4 hours.

LIMITS OF THE CURRENT SYSTEM

The current system as we have implemented it can handle approximately 2,500 runs per day. This is because there are a lot of input and output (I/O) operations and thus communication between the Rancher master node and the workers. Because we use an EBS volume in conjunction with each EC2 instance, there are limits on how long we can maintain the maximum throughput.

We have also imposed a limit of allowing a maximum of three worker nodes. This is largely because the TPC model runs at such a rate that by the time the run submission queue has created the 16th job, the first job has completed, and nothing would be gained by adding another worker to handle the 16th job—it simply takes the spot of the first job on an existing worker node. At present, we have not modified the TPC model source code much at all. If we were to speed up either by parallelizing the model itself or by undergoing a major refactoring of all the source code, these assumptions would change.

COSTS OF THE CURRENT SYSTEM

It does not cost even $0.02 per run in order to leverage the cloud. Cost breakdowns follow and assume a 24-hour running time on the "Always Running" pieces and a 4.5-hour running time on the "Max Run Submission" pieces.

		Price per hour
Always Running	Infrastructure for website and initial Rancher instance (t3.small & c5.large)	$0.1268
	EBS Volume ($12/month)	$0.0167
	Results storage (db.r3.xlarge)	$0.24
Max Run Submission (2,500 runs)	c5.large instance to send the runs	$0.085
	c5.4xlarge to process results	$0.68
	c5.9xlarge as worker node	$1.53
	c5.9xlarge as worker node	$1.53
	c5.9xlarge as worker node	$1.53
	EBS volumes for workers	$1.11
Total always on per day		$9.204
Total additional per max run		$26.2875
Grand total max run over 1 day		$35.4915

USAGE OF THE CURRENT SYSTEM

During 2018 and 2019, TPC has leveraged this cloud architecture to create an analysis of the Tax Cuts and Jobs Act of 2017[6] and an accompanying feature.[7] The team is now working on several other projects that require thousands of runs.

Aside from the current use of exploring a large combination of small changes in parameters, we see some other potential use cases for an architecture such as our current system. While the TPC model does not have an alignment procedure per se, other models at the Urban Institute do. This process typically requires running a model multiple times and checking against a target. By changing the behavior of the "create analysis file" step, one could find the best result for a particular target. Additionally, with a space of results to explore, one could imagine designing a search or other optimization function that would look for a result with a particular set of criteria.

LESSONS LEARNED

Based on our Modeling in the Cloud work, we offer the following lessons for those building and customizing cloud architecture.

AUTOBUILD PIPELINES THROUGH VERSIONING. How you build your pipeline depends on how you version your model. The TPC model is designed to have several versions available to run; other models only have one true production version. It worked well for our team to start small, figure this out, and then build out the cloud architecture.

UPDATE ALL COMPONENTS REGULARLY. Whatever tools you decide to use must be kept current. Schedule time to manage these updates and their implications. We spent considerable time upgrading Rancher and ensuring that all Docker images were still functioning properly post-update.

IDENTIFY POTENTIAL BOTTLENECKS. For our architecture, bottlenecks ended up being I/O operations, EBS volume limits, and Rancher overhead. When running many models, it is inevitable that something will break or not perform as well as expected. The key is learning how to trace such problems, design solutions, and test them. We have selected a General Purpose SSD (gp2) EBS volume volume to associate with each EC2 instance. Our current I/O operations' burst balance can run at maximum 3,000 IOPS for 1,862 seconds (31 minutes). Refilling the burst balance currently takes about 15 hours. It now takes approximately 2,500 runs to use up the entire balance. Runs of this size take about 4 hours from start to finish. A 15-hour wait time for the I/O burst balance to refill means we can only do one set of 2,500 runs per day.

DON'T OVEREXTEND YOUR AWS INSTANCES. At one point, we ran both the Rancher infrastructure and interface from the same AWS instance. As our memory needs increased, we found splitting these two functions was necessary.

MANAGE YOUR INPUTS. In particular, some more complicated input issues came to light when we started seeing nonpredictable failures in runs. For example, as noted, we moved our input files directly into the Docker images instead of keeping them separate in an S3 location for each job to access. In the earlier versions of our Docker container setup, all input files were stored on S3 and were copied into the containers at start using one of the helper containers in our stack. However, once we scaled up from tens of runs to hundreds, we were not able to copy files fast enough. This would cause jobs to sometimes bump into each other, resulting in one file (the exact file and the timing of the failure were unpredictable) not copying into the run container and causing job failure. To fix this, we added the input files to the Docker image to start. This increased the size of the image and meant our EC2 instance type that ran the models needed to be sized up, but we needed the stability.

ACCOUNT FOR MULTIPLE WAYS TO SCALE UP RUN VOLUME. Technology moves fast. If we were starting from scratch today, we probably would not use Rancher and Cattle. Rancher, we have learned, uses

considerable resources just to track the number of available worker nodes. Over the past few years, Kubernetes, another container management system, has become a more standard solution. In fact, the latest versions of Rancher use only Kubernetes. We have done some small proof-of-concept exercises with the Simple Tax Models using Kubernetes but have not experimented beyond that usage.

Other ways to expand might be to use tools and services like AWS Fargate, a tool that allows for the running of Containers as a Service instead of having to manage compute instances or by leveraging pySpark.[8]

BUILD FOR SCALE OR SCALE UP SLOWLY. Teams must constantly examine the trade-offs and cost benefits to speeding up the model itself versus using larger EC2 instance types. When creating a model from scratch, one can of course design it to run at this scale from the start (e.g., by using pySpark). When upgrading a model that for decades has been run in solo instances, to be run thousands of times at once, you must consider the inherent source code limits. Teams must decide which is more beneficial: modifying source code to remove those limitations, or spending a little more on compute and storage.

In our case, as one of our goals was to minimize model code changes, we moved slowly, first designing a system to run 10 to 50 runs at a time, then 100 to 500, and finally figuring out our maximum number of runs per day and examining the trade-offs in cost. We opted to use slightly more powerful compute options with enough memory to handle the TPC model as it was written. Were we to further optimize the model itself, we could likely reduce the size of our EC2 instance types and thus reduce compute costs. However, this could make the TPC model more difficult to maintain for the modeling team, and the cost of modifying the code could end up exceeding our savings on compute resources. Your team should consider this question carefully.

EXAMPLE CLOUD-BASED MODEL: THE SIMPLE TAX MODEL

Because we cannot share the full TPC model or our entire workflow, we will instead share a very simple tax model and use it to illustrate setup details. The model code described here can be downloaded from https://github.com/UI-Research/Simple_Tax_Models and is provided in three languages: Fortran, C++, and Python. We will use the Fortran version in our examples here. We assume readers will have a basic understanding of how to create and access AWS EC2 instances. More information on this topic is available at https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/get-set-up-for-amazon-ec2.html.

MODEL OVERVIEW

Regardless of language, each version of the Simple Tax Model does the same thing. It reads an input file, does a straightforward mathematical calculation, and outputs a microdata file. Then a Python script processes that microdata file to produce summary tables.

The input file contains the following:

- `itemized`: a double value with the amount of itemized deductions for the unit
- `salary`: a double value with the amount of salary for the unit
- `filingStatus`: the filing status of the unit; 0 indicates single and 1 indicates married
- `weight`: a double value with the weight of the unit

The original data source is the 2010 Survey of Consumer Finances and can be downloaded at https://www.federalreserve.gov/econres/scf_2010.htm. Before processing, and as shown in the following SAS code, all units with 0 salary have been removed. Input data and the SAS code used to generate the file are included in the GitHub repository.

```
libname in 'D:\SCF10';

data temp; set in.p10i6;
        keep IDN Schedule_A Salary Filing_stat wgt;

        if x5744=2 then delete;
        if x5746 in (2,4,5) then delete;
        IDN=y1;

        Schedule_A=x5823;
        Salary=x4112+x4712;
        if salary=0 then delete;

        Filing_stat=(x5746=1);

        wgt=x42001/5;
```

```
run;

proc univariate;
var schedule_a;
weight wgt;
where schedule_a>0;
run;

PROC EXPORT DATA= WORK.TEMP
            OUTFILE= "D:\Demo.csv"
            DBMS=CSV REPLACE; PUTNAMES=YES;
RUN;
```

The parameters defined in the model are as follows:

- `RATE`: a decimal value indicating the tax rate (e.g., 0.1 = 10%)
- `STANDARD`: a double value indicating the amount for the standard deduction
- `EXEMPT`: a double value indicating the amount for a personal exemption. The calculation for a standard deduction of $1,500 would be to choose `max(itemized, 1500)`.

A simple parameter file used to run the model would look like:

```
RATE,0.1
STANDARD,1500
EXEMPT,1000
```

The calculation of taxes in the model is contained in the file `TaxFiler.f90` and is simply:

```
module TaxFiler
use declarations

implicit none
save

contains

  ! Description: Calculates tax in a very simple manner.
  !
  ! Remarks: Everything is globally declared in declarations,
  !   so no need to pass parameters.
  subroutine calcTax
    tax = rate *(salary - max(itemized, standard) - 1000 * (1 + filingStatus));
  end subroutine

end module
```

The output file (`output.csv`) contains all of the input variables and appends this tax calculation as an output to the end of the file.

The Python code below, `summarize.py`, produces final summary tables of the output. This is run in conjunction with every version of the simulation regardless of programming language used.

```
#python3
#
# Jessica A. Kelly
# The Urban Institute
# jkelly@urban.org

# example usage: python3 create_summary_tables.py ../CPP/output.csv

#import csv
import pandas as pd
import os
import argparse
import sys
import csv


# set up arguments to send at command line.
# input is a required argument. if need optional, example is below
def create_arg_parser():
    """Creates and returns the ArgumentParser object."""

    parser = argparse.ArgumentParser(description='Description of your app.')
    parser.add_argument('input',
                    help='Path to the input file.')
    #parser.add_argument('--outputDirectory',
    #                help='Path to the output.')
    return parser

# load data into dataframe
def load_data(file_path):
    df = pd.read_csv(file_path)
    return df

# calculate salary bins and return the binned results
def group_by_salary(df):
    #group continuous salary into defined bins
    levels = [min(df["salary"]), 400, 7000, 50000, 93000, max(df["salary"])]
    labels = ["salary <= 400", "400 < salary <= 7,000", "7,000 < salary <= 50,000",
            "50,000 < salary <= 93,000", "salary > 93,000"]
    return pd.cut(df["salary"], bins=levels, labels=labels, include_lowest = True)



# main function
if __name__ == "__main__":
    arg_parser = create_arg_parser()
    parsed_args = arg_parser.parse_args(sys.argv[1:])
    if os.path.exists(parsed_args.input):
       PATH = parsed_args.input
```

```
    # load and do some initial calculations we need to use later
    df = load_data(PATH)
    df["wgtTax"] = df["tax"] * df["weight"]
    df["avgTax"] = df["tax"] / df["salary"]

    # total tax revenue
    totalTaxRev = df["wgtTax"].sum()
    #print(totalTaxRev)

    # group continuous salary into defined bins
    df["binned"] = group_by_salary(df)
    # and then add up average tax over the bins
    dfavg = df["avgTax"].groupby(df.binned).sum()
    #print(dfavg)

    # then the top decile >140000
    top_decile = df.loc[df["salary"] > 14000, "avgTax"].sum()
    #print(top_decile)
    # and top one percent, over 475000
    top_one_perc = df.loc[df["salary"] > 475000, "avgTax"].sum()
    #print (top_one_perc)

    results = {"total tax revenue": totalTaxRev,
        "salary < 400": dfavg[0],
        "400 < salary <= 7,000": dfavg[1],
        "7,000 < salary <= 50,000": dfavg[2],
        "50,000 < salary <= 93,000": dfavg[3],
        "salary > 93,000": dfavg[4],
        "top decile": top_decile,
        "top one percent": top_one_perc}

    pd.DataFrame.from_dict(data=results, orient='index').to_csv('summary_output.csv',
header=False)
```

The summary tables produced calculate total revenue as the sum of the calculated tax multiplied by the weight across the entire file. The code then produces a table showing the sum of tax divided by salary using groupings by salary. This functionality is adapted in the Lambda functions that are used in the following walkthrough.

PROGRAMMATICALLY GENERATING PARAMETER FILES

A revised version of the full ParamUpdater module is provided in the parameters folder. An example implementation file, create_param_files.py, is also provided in the same folder. It shows several different ways of updating a parameter file.

A SIMPLE WORKFLOW TO RUN 20 TIMES

Next, we lay out the steps to create a simple stack in Rancher, generate parameter files, and create an analysis dataset from 20 runs of the Simple Tax Model. To follow along, fork the Simple_Tax_Model repository to your GitHub user account. While we will not replicate the full workflow as implemented for the TPC model, we will illustrate the important concepts.

STEP 0: PREPARE PARAMETER AND OTHER HELPER FILES. Using the ParamUpdater, create a set of 10 to 20 parameter files. Save these to an AWS S3 location. Note the link to the location, and ensure that the bucket is available for the EC2 instances we create later. To follow along, you will need to have made edits to other files that reference the S3 location in order to load the parameter files into the container that runs the model.

Update the Python file `run-rancher-compose.py` to contain the names of your parameter files:

```
    submission =
{"parameter_list":[{"Key":"parameterOptions_Standard_1600.csv"},{"Key":"parameterOptions_St
andard_1700.csv"},{"Key":"parameterOptions_Standard_1800.csv"}]}
```

Note that this is a manual step that could and should be replaced by code. In the full TPC workflow we use a Lambda function to get this information.

Next, you will need to edit the image attribute in the `docker-compose-job-submitter-template.yml` file to point to your DockerHub account, as shown in this `docker-compose` file:

```
version: '2'
services:
  runmodel:
    image: yourdockerhubaccount/simple_tax_models
    stdin_open: true
    tty: true
    labels:
      io.rancher.container.start_once: 'true'
      io.rancher.container.pull_image: always
      io.rancher.scheduler.global: 'true'
    command: [ "{PARAMETER_FILE}","{OUTPUT_FILE}" ]
```

Once edited, copy the files `docker-compose-job-submitter-template.yml` and `run-rancher-compose.py` to the same S3 location as your parameter files.

The last thing to upload to this S3 location is the file `summarize_lambda.zip`. We will use this set of files when we create our AWS Lambda function to generate summary tables in step 4.

STEP 1: GITHUB AND CREATING A DOCKERFILE. We use a container to define our runtime environment, and a small shell script to submit jobs. The Dockerfile for the Fortran example is shown below:

```
# Simple_Tax_Public_Model
# Fortran version

# start by building the basic container
```

```
FROM centos:latest
MAINTAINER Jessica Kelly <jkelly@urban.org>
RUN yum update -y
# add fortran
RUN yum install -y gcc-gfortran gdb make
# and python and related requirements
RUN yum -y install epel-release && yum clean all
RUN yum -y install python-pip && yum clean all
RUN pip install awscli

ENV REGION=us-east-1

# build the model code - note the copy from the parent directory of the bash script and
requirements
COPY Fortran/Makefile run_fortran.sh requirements.txt Fortran/*.f90
Fortran/parameterOptions.csv /Fortran/
COPY data/Demo.csv /data/

# set working directory to Fortran and then build the model
WORKDIR /Fortran/
#add python requirements
RUN pip install -r requirements.txt
RUN make

# configure the container to run the executable by default
# CMD ["./fortmodel"] #this is simplest way to run the exe, but need more steps so use bash
ENTRYPOINT ["/Fortran/run_fortran.sh"]
```

A few things to note. First, we have opted for readability. There are likely ways to make this file shorter. Second, note that the input data is read into the container here (COPY data/Demo.csv /data/) We can do this because it is small. We also do this to avoid any issues with read locks later in processing.

The bash script that is called when the model is run is shown below:

```
#!/bin/bash
# expects two arguments, first for parameter file, second for output file
# copy the parameter file
FILE="path/to/your/bucket/$1"
echo "Attempting download : $FILE"
# copy parameter file
aws s3 cp $FILE $1
#done

#need a little tweak to output file name
OUTNAME="$2.csv"
echo "running fortran"
./fortmodel "$1" "$OUTNAME"
echo "fortran complete"

# copy results
OUTFILE="path/to/your/out/bucket/$2.csv"
aws s3 cp $OUTNAME $OUTFILE
```

Note here that you will need to specify the AWS S3 location of your parameter files and a separate AWS S3 location for your output files. We copy the parameter file for a single run into the container in this step. When we execute, we will give the name of a single parameter file in the command.

STEP 2: AUTOMATING THE DOCKER BUILD WITH DOCKERHUB. Next, we need to place the Docker image we just defined where it can be accessed when needed. [9] In this example, we will forgo any complicated options, and as simply as possible, create an autobuild connected to a GitHub repository.

First, create and connect a DockerHub account to your GitHub account in order to create a link between the two components:

1. Create a DockerHub account and log in.
2. Navigate to **My Profile > Settings > Linked Accounts & Services**.
3. Select **Private connection type** for the automatic builds and enter the login credentials for your GitHub account.

Next, create an Automated Build from your forked version of the Simple_Tax_Models GitHub repository:

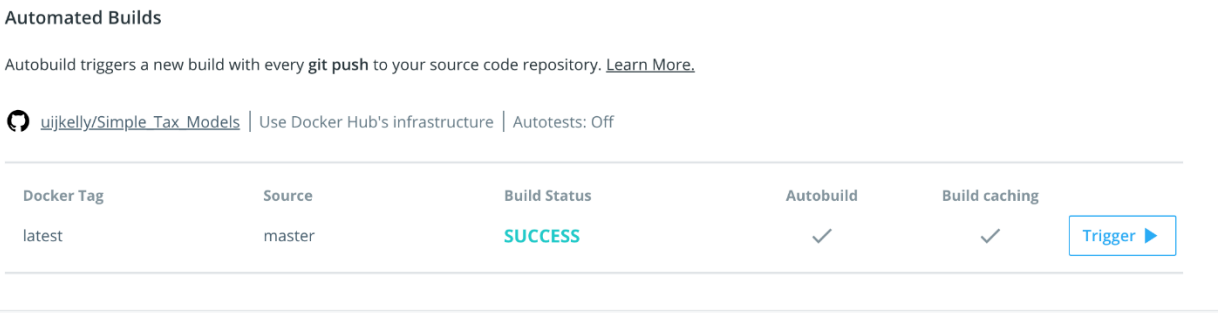
1. In the repository, rename the Dockerfile of your choice (`Dockerfile_fortran`, `Dockerfile_python`, or `Dockerfile_cpp`) to simply `Dockerfile`. `DockerHub` assumes the `Dockerfile` lives at the root of your source and pushes the changes on GitHub.
2. Create a new DockerHub repository connected to GitHub. On DockerHub, click the **Create Repository** button, enter a name, and in the **Build Settings** section, click the **GitHub** icon. Then select the appropriate GitHub repository.
3. Click **Save and Build**. The system displays the home page for the `Simple_Tax_Models` Automated Build.

To manually initiate a build, simply click **Trigger**, and the automated build process will begin. Additionally, any time a change is made to the master branch, a new build will be triggered.

Figure 3 below shows a DockerHub screen indicating a successful connection and build of the GitHub repository.

FIGURE 3

Successful Connection to GitHub and Successful Build on DockerHub



For this simple example, we will skip the step of creating any webhooks from our full workflow. This step becomes more important when multiple versions of a model need to be available at a given time; this behavior is highly dependent on use-case. Webhooks in both DockerHub and GitHub are well-documented processes, and links are available in the "Resources" section.

STEP 3: CREATE AND CONFIGURE RANCHER INFRASTRUCTURE. In this step we will walk through creating two AWS EC2 instances and installing and configuring Rancher and Docker on the instances. One EC2 instance will run Rancher, and one will be a worker node to run the Simple Tax Model. We also ensure that DockerHub is properly connected and configured.

Full Rancher version 1.6 instructions are available here: https://rancher.com/docs/rancher/v1.6/en/installing-rancher/installing-server/.

STEP 3A: CREATE AWS INSTANCES. First, instantiate an EC2 instance for the Rancher infrastructure. For illustrative purposes, we will use a fairly small instance. In production, this should be properly sized to meet your needs. For more information about EC2 instances, see the AWS documentation: https://aws.amazon.com/ec2/instance-types/.

Resource Type	Value
Amazon OS Image	64 bit Amazon Linux AMI
AWS Instance Type	T2.small
Region	US East
vCPU	1
Memory	2 GiB
Elastic IP	Will vary; in this example we will use 192.0.0.0

As you create this instance, be sure that you configure to allow for ssh and allow port 8080 for inbound TCP. For more information about setting up EC2 instances, see the AWS documentation:

https://docs.aws.amazon.com/efs/latest/ug/gs-step-one-create-ec2-resources.html. Also note the IP address, as you will need this in subsequent steps.

Next, add one worker node for the Simple Tax Model. In general, for our production purposes, we have one worker node that is always running. Additional nodes are added only as they are needed.

Instantiate a new EC2 instance for use as a worker node. As with the Rancher infrastructure instance, we are using a very small instance in this example because it is all that we need.

Resource Type	Value
Amazon OS Image	64 bit Amazon Linux AMI
AWS Instance Type	T2.small
Region	US East
vCPU	1
Memory	2 GiB
Elastic IP	Will vary; in this example we will use 192.0.0.0.1

If you want to do things without making everything public, you should also assign an IAM Role. You will use this role in other steps to ensure that all of the AWS components, from AWS S3, EC2, and Lambda, can communicate with one another properly. Links to AWS documentation on this topic are in the "Resources" section.

STEP 3B: INSTALL AND CONFIGURE RANCHER AND DOCKER. Once the EC2 instances are created and running, ssh into your Rancher infrastructure instance and install Docker. The following commands can be used after you ssh into the AWS instance. [10]

```
$ sudo yum update -y
$ sudo yum install -y docker
$ sudo service docker start
```

You can check the Docker status at any point by using the command:

```
$ sudo service docker status
```

If the Docker service is inactive or not running (at any point), use the commands:

```
$ sudo service docker enable
$ sudo service docker start
```

Then, once Docker is verified to be active, install Rancher using the command:

```
$ sudo docker run -d –restart=unless-stopped -p 8080:8080 rancher/server
```

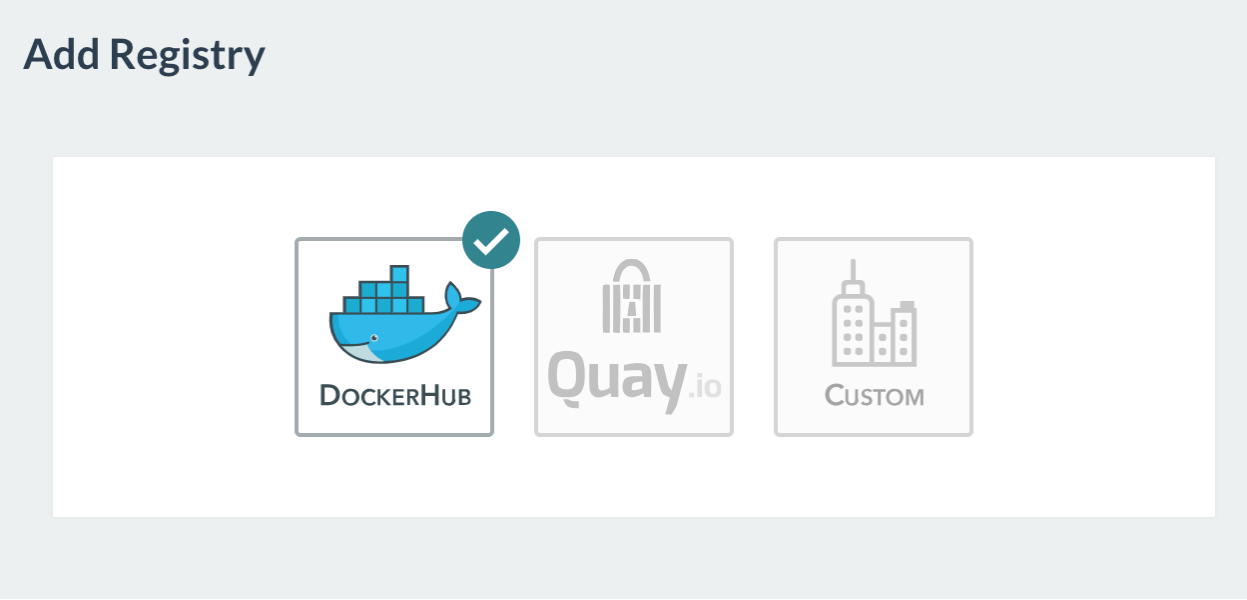
Once the installation is done, try to access Rancher via the web interface it installed. This link will look like http://192.0.0.0.0:8080, where the IP address is the elastic IP of your EC2 instance.

For our purposes in this tutorial, it is fine to use **Local Auth**. Security is another area you should customize during production for your application. Integrating with Active Directory or a Single Sign On utility is also possible.

The next step to configure the Rancher installation is to integrate with DockerHub. To do this, in the top menu, select **Infrastructure > Registries**. Ensure that **DockerHub** is selected as shown below, and then enter your DockerHub credentials and click **Create** to complete.

FIGURE 4

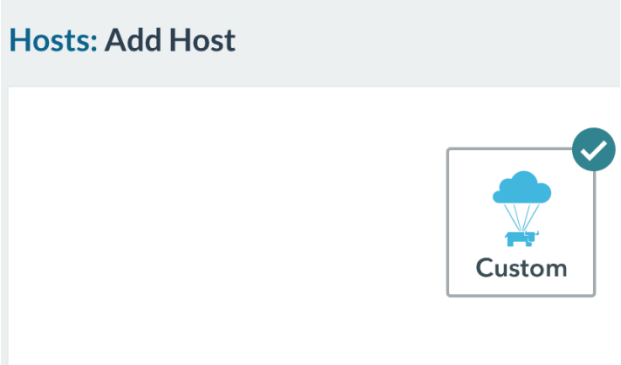
Select DockerHub When Configuring Rancher



Now to configure the worker node. As with the Rancher infrastructure instance, ssh into the worker node EC2 instance and install Docker as shown at the start of this step.

Next, add a Rancher Host to point to your worker node. Ensure that your new Environment is still chosen/active, select **Infrastructure** from the top Rancher menu, click **Hosts**, then click the **Add Hosts** button. Then ensure that the custom machine driver is selected as shown below.

FIGURE 5
Select Custom Machine Driver When Configuring Hosts

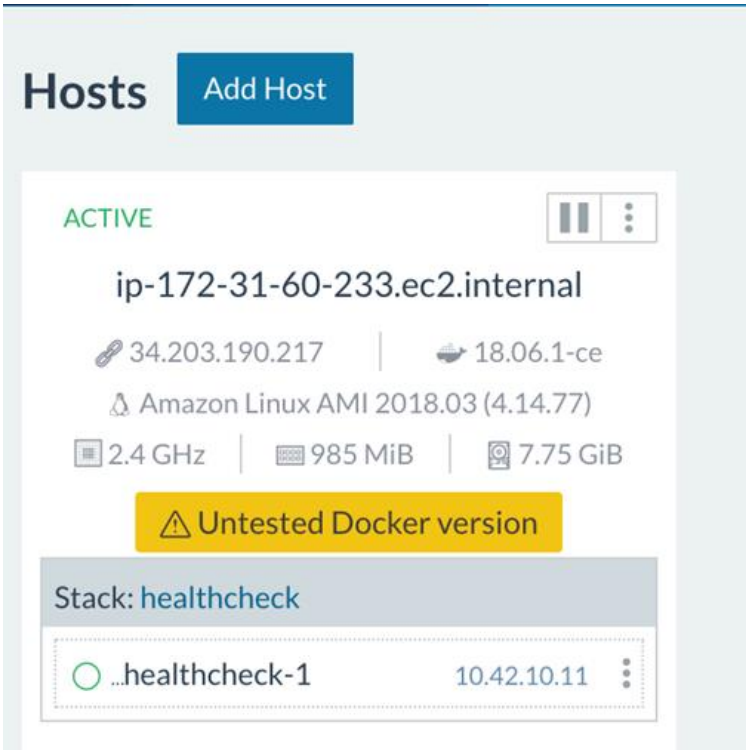


Next, copy the Docker command that is shown in the interface. Use ssh to enter the worker node 192.0.0.0.1, and paste in and execute the command from Rancher to register the host. This command will look something like:

```
$ sudo docker run --rm --privileged -v /var/run/docker.sock:/var/run/docker.sock -v
/var/lib/rancher:/var/lib/rancher rancher/agent:v1.2.10 http://
192.0.0.0.0:8080/v1/scripts/<registration-token>
```

The new host should appear shortly and look something like the example in figure 6 below.

FIGURE 6
Successfully Created Rancher Host



Note that many of these steps can be automated and refined (and have been in our full implementation of the TPC model), depending on the specific use case.

Last, you will need to create a file called credentials.py. This should contain your `RANCHER_URL`, `RANCHER_ACCESS_KEY`, and `RANCHER_SECRET_KEY`:

```
RANCHER_ACCESS_KEY = '<access-key>'
RANCHER_SECRET_KEY = '<secret-key>'
RANCHER_URL = 'http://rancher-ip:8080'
```

To generate your `RANCHER_ACCESS_KEY` and `RANCHER_SECRET_KEY`, navigate to API in the top Rancher menu, click **Keys**, and then **Add Environment API Key**. Copy this new `credentials.py` to the same S3 location as the other files. **Note that you should not add this file to any GitHub repositories as it will contain confidential information.**

For this simple example, we will skip creating any additional worker nodes. This could be done manually in the same manner as you created the first worker node. To do this using code, see the above snippet in "Queue Creation, Spinning Up/Down Workers," the documentation for the Rancher command line interface (https://rancher.com/docs/rancher/v1.6/en/cli/), and the documentation for the boto3 library (https://boto3.amazonaws.com/v1/documentation/api/latest/index.html). Our model won't need an additional worker node to handle run submission. A load balancer is generally also configured at this step to ensure smooth handling of runs to multiple workers.

STEP 3C: SETTING UP RANCHER-COMPOSE. Next, we will install Python and `rancher-compose`, and copy in the supporting files edited and saved in steps 3A and 3B. [11] Edit the shell script `Rancher-Compose.sh` provided in the `Simple_Tax_Models` repository to reference your S3 location.[12] Use a tool like `scp` to copy the file into the AWS EC2 Rancher Infrastructure instance, and then once you ssh into the instance, use the commands below to execute the script. The files copied will be used as we submit model runs.

```
$ chmod +x Rancher-Compose.sh
$ sudo ./Rancher-Compose.sh
```

Once the script completes, you should have several files in your directory `/usr/local/sbin`.

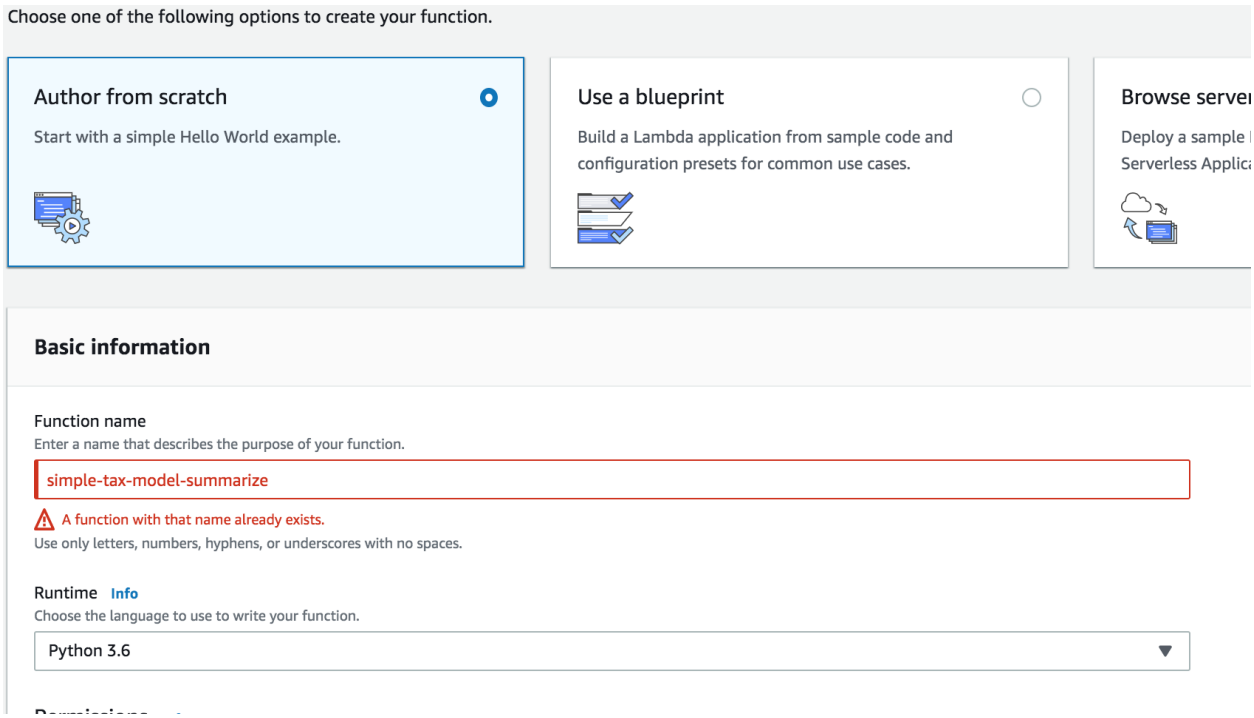
STEP 4: SET UP THE ANALYSIS TRIGGER. Note that this flow is different from the finalized TPC model flow. Because analysis can be quite different across models, and because we do not set up a database for this simple illustration, we will simply define an AWS Lambda function that will run once files are copied into a specific S3 location. This model is in fact simple enough that the analysis step could be done from within model processing itself. We have opted to add a Lambda function to illustrate the concept. You will need three S3 buckets: one for parameter files and other files to copy to EC2 instances, one for output of model results, and one final location for the summary results. We will refer to the first bucket here and in code samples below as simple-tax-model, the output bucket as simple-tax-model-output, and the summary-results bucket as simple-tax-model-summary.[13]

We will also need to configure our AWS Lambda function to operate properly. The Python code that summarizes the microdata leverages the pandas library. Because this is not a part of the standard Lambda environment, we needed to add it. To do so, we have compiled the library and provided it in this repository as a ZIP file. [14] This file also contains the Lambda function itself. Edits to the function should not be required. In step 0, you should have uploaded summarize_lambda.zip to the simple-tax-model S3 location. If you have not, do so now.

Create a new Lambda function with the Python 3.6 runtime on the AWS Lambda Dashboard by clicking the **New Function** button as shown below in figure 7.

FIGURE 7

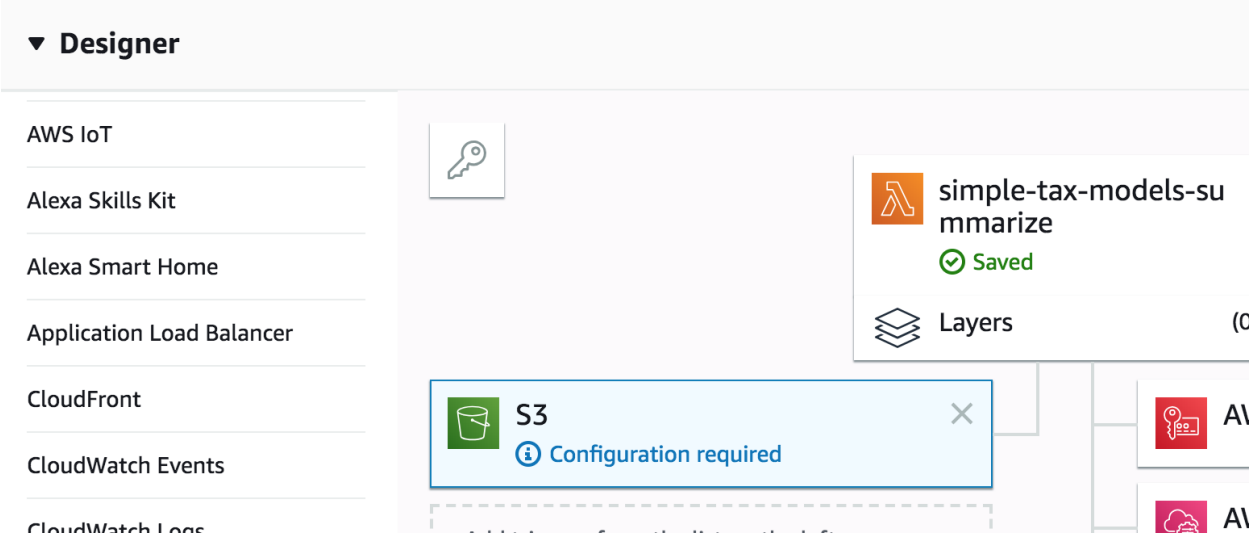
AWS Lambda Function Creation



When creating a new Lambda function, ensure that you have created an IAM role that will allow you to access the `simple-tax-model-output` bucket. See the "Resources" section for more information on this process.

Click the button to create and you will then be taken to the Configuration page. In the **Designer** dialog, choose **S3** as the trigger as shown in figure 8.

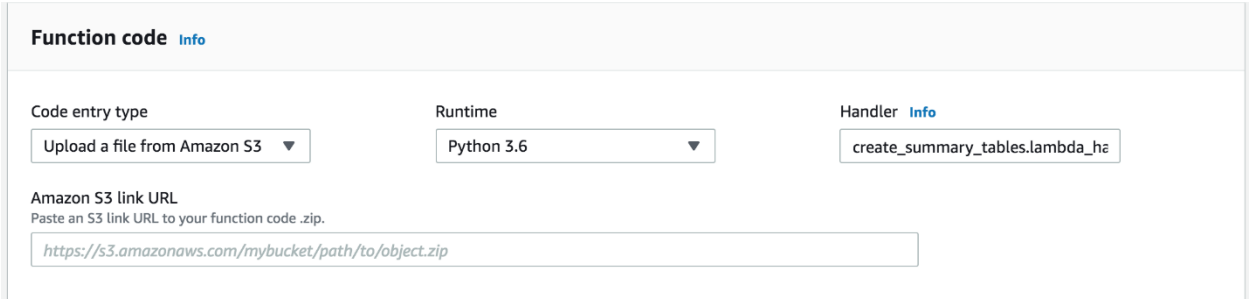
FIGURE 8
Lambda Designer Dialog with S3 Selected as Trigger



And configure to use your `simple-tax-model-out` bucket. You can further specify **All object create events** to trigger running the function, and filter so that only CSV files trigger the function.

Next, we will load and specify the ZIP file in the **Function code** section of our Lambda function. Note the name in the **Handler** box and the **Code Entry Type** as shown in figure 9.

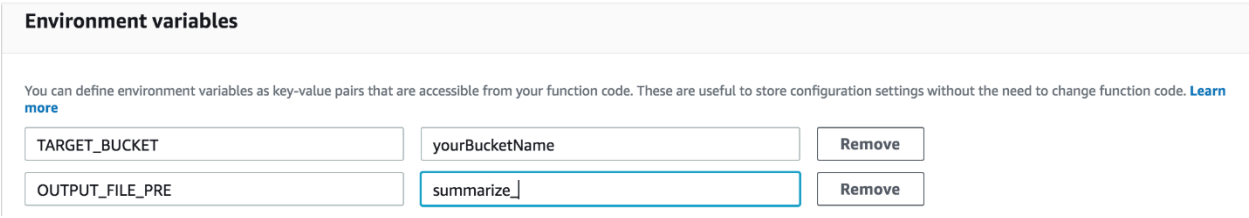
FIGURE 9
Specifying Lambda Function Code



Last, we add two environment variables to control where the output is written. These keys are called `TARGET_BUCKET` and `OUTPUT_FILE_PRE` and should be specified to point to your bucket name for `simple-tax-model-summary` and a prefix to use for the files. The exact values will depend on your final S3 location for the summarized output.[15] Figure 10 below shows example placeholder values.

FIGURE 10

Setting Environment Variables for Lambda



STEP 5 TEST THE LAMBDA FUNCTION. Copy a test output file (provided in the GitHub repository at `tests/output.csv`) to the `simple-tax-model-output` S3 location. The Lambda function should run and you should shortly see a new file called `_summarize.csv` in the S3 location `simple-tax-model-summarize`.

STEP 6: RUN THE MODEL 10 TO 20 TIMES. Here we will simply execute a Python script from the instance running the main Rancher Infrastructure, copy the output files to an S3 location, and trigger the analysis code.

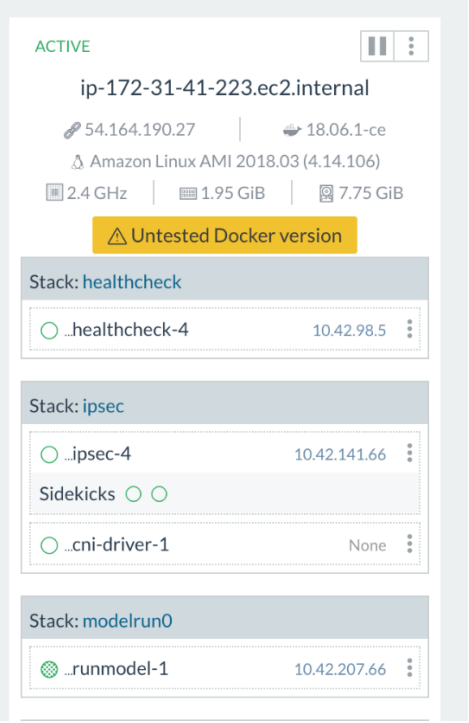
To run the model, revise the file `run-rancher-compose.py` provided in the `Simple_Tax_Models` repository to loop over your parameter files and generate a call to `rancher-compose`. To run, ssh into this EC2 instance, and issue the command below, modifying the path(s) as needed:[16]

```
$ python3 run-rancher-compose.py path/to/docker-compose-job-submitter-template.yml
```

You should be able to see the runs starting and running by looking at the hosts running in Rancher, as shown in figure 11 below. In Rancher, click on **Infrastructure** in the top menu and then select **Hosts**.

FIGURE 11

Running Rancher Host with modelrun Stack



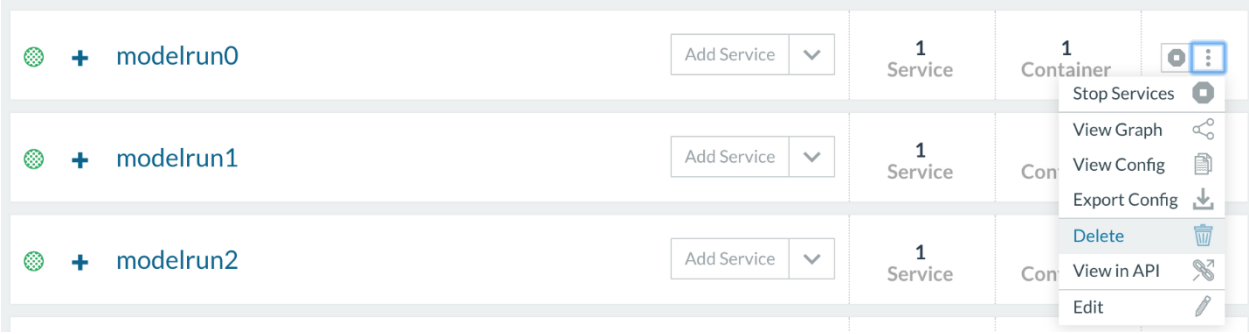
To view the log of a running model, click on the **vertical ellipsis icon** and select **View logs**. The example below shows the model running:

```
4/5/2019 9:36:45 AM  running fortran
4/5/2019 9:36:45 AM   Fortran-based Simple Tax Public Model Example
4/5/2019 9:36:45 AM   Number of Arguments is            2
4/5/2019 9:36:45 AM         10000  observations processed.
4/5/2019 9:36:46 AM         20000  observations processed.
4/5/2019 9:36:46 AM         20017  total observations processed.
4/5/2019 9:36:46 AM  fortran complete
4/5/2019 9:36:46 AM  Completed 256.0 KiB/2.8 MiB (3.0 MiB/s) with 1 file(s) remaining
```

If you need to rerun, you will need to manually delete the `modelrun` stacks in Rancher before doing so. To get to this screen, select **Infrastructure** on the top menu, then **Containers**. For the container you wish to delete, select the **vertical ellipsis icon**, then select **Delete** as shown in figure 12.

FIGURE 12

Deleting a Container in Rancher



STEP 7: VIEW THE OUTPUT AND THE ANALYSIS FILES. Once the models have completed running, navigate to your S3 locations to view and download microdata and summary results files. This step can also be further automated: for example, an email with links could be generated.

STEP 8: FINAL CLEANUP. At the end of processing, remove all worker nodes. This can also be automated, but it is simple to stop and remove worker nodes manually inside the Rancher interface. When runs have completed, select the Host in Rancher, deactivate and delete the host, then stop and delete the AWS EC2 instances. Additionally, the Rancher infrastructure can be taken down as well if it is no longer of use. As shown in the preparation steps 3A- 3C above, configuring Rancher is not a trivial amount of work, so keeping it up while experimenting saves time.

COMPARISON TO FULL WORKFLOW

Note that in the full TPC workflow, we have a separate queue instance to do this work, and a separate instance that holds a user interface, which both are used to define run submission. This is a highly customized part of the workflow and should be implemented in production to suit your needs.

A few important things to note:

- AWS Lambda functions are charged on a per-run basis. Since we are only running 20 in this example, this is not a huge concern, but when running thousands of models, it is. The function to create the analysis could be revised to be run just once, when all runs are complete;
- we manually specify the parameter file names, and generate output file names simply by adding an integer to the end of the string modelRun. In the full workflow, the parameter names are given via the interface; output filenames are generated using a hash on current time, and also kept track of as they relate to the inputs in a database. We have removed all this complexity from the Simple Tax Model implementation.

- the speed issues we faced at this step in the full TPC workflow were solved by implementing multiprocessing of our analysis files. Here we opt for straightforward calculation;
- in the full TPC workflow, we roll up the analysis even further, but skip this step in the example. All of these concerns are specific to what kind of analysis is desired.

RESOURCES

AMAZON WEB SERVICES

AWS Glossary https://docs.aws.amazon.com/general/latest/gr/glos-chap.html

AWS User Guide for EC2 https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/get-set-up-for-amazon-ec2.html

AWS Fargate https://aws.amazon.com/fargate/

AWS gp2 volumes and throughput https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/EBSVolumeTypes.html

AWS IAM https://docs.aws.amazon.com/IAM/latest/UserGuide/id_roles.html

AWS IAM roles and EC2 https://docs.aws.amazon.com/IAM/latest/UserGuide/id_roles_use_switch-role-ec2.html

AWS IAM roles and Lambda https://docs.aws.amazon.com/lambda/latest/dg/access-control-identity-based.html

AWS EC2 https://aws.amazon.com/ec2/instance-types/

AWS EC2 Tutorial https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-tutorials.html

AWS S3 https://docs.aws.amazon.com/AmazonS3/latest/dev/Welcome.html

AWS Lambda https://docs.aws.amazon.com/lambda/latest/dg/welcome.html

GIT AND GITHUB

Git https://git-scm.com/

GitHub https://github.com/

GitHub webhooks https://developer.github.com/webhooks/

DOCKER AND DOCKERHUB

Docker https://www.docker.com/

Docker Compose https://docs.docker.com/compose/

DockerHub https://hub.docker.com/

RESOURCES

Docker overview https://docs.docker.com/engine/docker-overview/

Docker webhooks https://docs.docker.com/docker-hub/webhooks/

RANCHER

v1.6 documentation https://rancher.com/docs/rancher/v1.6/en/

v2.0 documentation https://rancher.com/docs/rancher/v2.x/en/

Rancher Compose https://rancher.com/docs/rancher/v1.6/en/cattle/rancher-compose/

Rancher API Keys https://rancher.com/docs/rancher/v1.6/en/api/v2-beta/api-keys/

Rancher Command Line Interface https://rancher.com/docs/rancher/v1.6/en/cli/

OTHER

JSON https://www.json.org/

Kubernetes https://kubernetes.io/

R and RShiny: https://www.r-project.org/ and https://shiny.rstudio.com/

boto3 https://boto3.amazonaws.com/v1/documentation/api/latest/index.html

pySpark https://spark.apache.org/docs/0.9.0/python-programming-guide.html

ABOUT THE AUTHORS

Jessica A. Kelly is the director of research programming in the Office of Technology and Data Science at the Urban Institute. Kelly leads the Modeling in the Cloud initiative. She focuses on research projects that leverage the microsimulation models developed at Urban, working to expand and scale the models to handle both larger data and in running simultaneous runs. She holds a bachelor's degree in mathematics from Towson University.

Kyle Ueyama is a senior programmer and data scientist in the Office of Technology and Data Science at the Urban Institute, where he works on a wide range of projects, such as the Modeling in the Cloud initiative. He is an experienced programmer in R and Python, and he helps lead the users' groups for both languages. He holds a master's degree in quantitative methods from Columbia University and a bachelor's degree in political science from the University of California, Los Angeles.

Alyssa Harris is a former programmer and analyst on the research programming team in theOffice of Technology and Data Science at the Urban Institute. She worked with researchers to maintain and enhance microsimulation models, such as the Transfer Income Model, and to translate policies and research principles into code. Harris is now a cloud and data engineer at Development Seed. Harris holds a BS in computer science and a BA in economics and mathematics from Rhodes College, and a master's degree in economics from the University of Virginia.

[1] In simple terms, this is just a way for one web-based application to communicate with another web-based application.

[2] See the Rancher v1.6 documentation here: https://rancher.com/docs/rancher/v1.6/en/.

[3] See "Manage Your Inputs" under "Lessons Learned" later for details. We changed from loading the input files via a copy command to including them in the base image as we progressed in this project.

[4] We have imposed limits of 2,000 runs at a time to manage costs and to ensure that we maintain a consistent throughput. We discuss this limitation further in "Identify Potential Bottlenecks" under "Lessons Learned".

[5] For more information about how to implement multiprocessing in Python, see https://medium.com/@urban_institute/using-multiprocessing-to-make-python-code-faster-23ea5ef996ba

[6] Robert McClelland, Daniel Berger, Alyssa Harris, Chenxi Lu, and Kyle Ueyama, "The TCJA: What Might Have Been," Washington, DC: Urban-Brookings Tax Policy Center. March 26, 2019, https://www.taxpolicycenter.org/publications/tcja-what-might-have-been.

[7] Robert McClelland, Daniel Berger, Alyssa Harris, Chenxi Lu, Kyle Ueyama, and Jessica Kelly, "Exploring Alternatives to the Tax Cuts and Jobs Act," Washington, DC: Urban-Brookings Tax Policy Center. March 26, 2019, https://apps.urban.org/features/tax-cuts-and-jobs-act-alternatives/.

[8] We have been testing pySpark as an orchestrator for models written solely in Python, and we see promise in leveraging other languages this way as well.

[9] Docker provides a deeper tutorial, https://docs.docker.com/docker-hub/builds/#create-an-automated-build, that explains more advanced setups and features. In particular, for a complex project, consider the autobuild tags.

[10] You may have to create or assign superuser status to your `ec2user` or run at root level. For more information, see the AWS portion of the "Resources" section.

[11] For more information about creating compose files, see the links in the "Resources" section. We will use these files as we programmatically call `rancher-compose` to create a service to run the simple model. Rancher runs whenever a service is

created. You can see this operation yourself by creating a new service, specifying the `docker-compose.yml` and `rancher-compose.yml` files, and seeing the model run.

[12] We have found that using `scp` to copy more than one file can be problematic as permissions are adjusted. Thus, we keep files in AWS S3 and copy into the EC2 instance. Copying between AWS services is fast and free.

[13] We need this third location because the Lambda function will trigger on a file being created in the `simple-tax-model-output` S3 location. If we were to write the summary results to this same place, we would create an infinite loop.

[14] See https://hackersandslackers.com/using-pandas-with-aws-lambda/ for a full walkthrough of the steps.

[15] As a second example, if your summary S3 location is `mybucket/summaryoutput` then `TARGET_BUCKET = mybucket` and `OUTPUT_FILE_PRE = summaryoutput/summary_`

[16] These files will all copy to `/usr/local/sbin` and will require superuser or root-level permissions to run at the command line.

The Tax Policy Center is a joint venture of the
Urban Institute and Brookings Institution.

BROOKINGS

For more information, visit taxpolicycenter.org
or email info@taxpolicycenter.org