Covalent

ResearchOps

Challenges and Practical Solutions for Distributed Scientific Computing

Will Cunningham, PhD
Head of Software
Agnostiq

Venkat Bala, PhD
HPC Engineer
Agnostiq

Overview.

1. Introduction to ResearchOps – Computational Research Operations
2. Covalent – Workflow orchestration tool for quantum + HPC
3. Code Examples
4. HPC in the Cloud (and beyond!)
5. Hands-On Tutorial – Molecular Dynamics

Computational Research Operations

Why is computational research hard?

Primary Challenges:

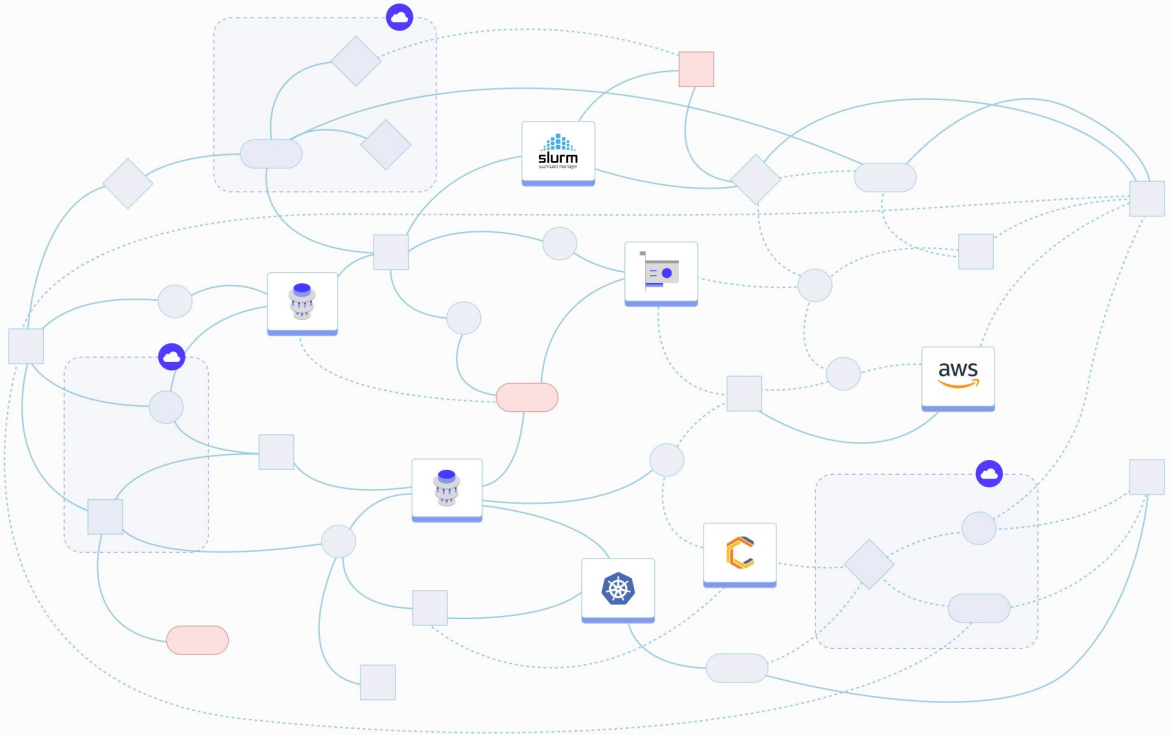
- Explore new research problems
- Apply interdisciplinary techniques
- Push the boundaries of hardware
- Find hidden patterns in data

Additional Operational Challenges:

- Organizing and versioning experiments
- Managing and sharing data
- Managing accounts on multiple clusters
- Checkpointing long simulations
- Reproducing (your own) work
- Package version conflicts
- Compiling colleagues' code
- Incorporating legacy code
- Identifying optimal hardware resources
- Infrastructure management
- Working within a budget constraint

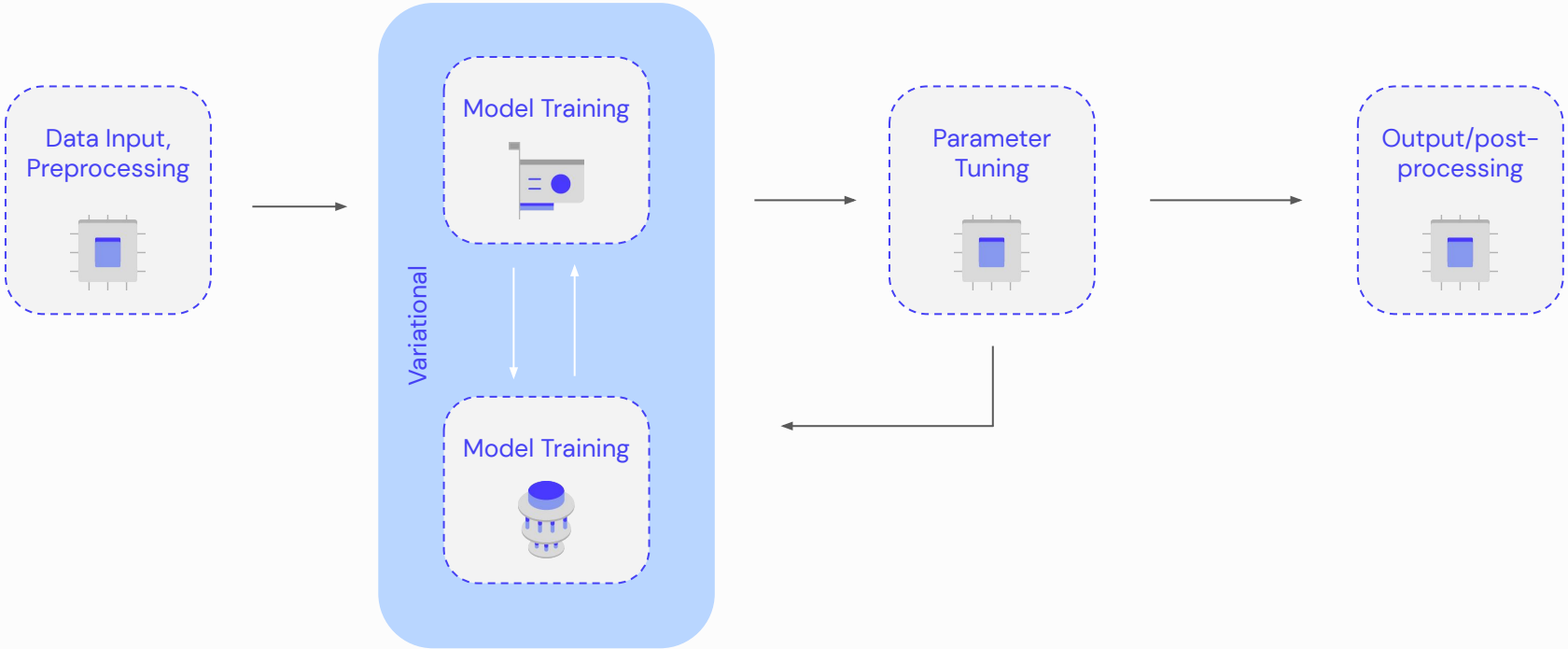
Lots of work, little reward... everyone comes up with their own bespoke solutions to these

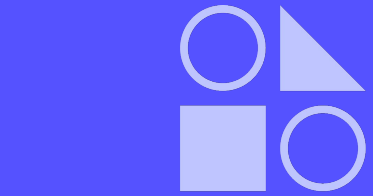
Modern computational workflows are complex.



Quantum is driving the need for more heterogeneity.

A simplified quantum machine learning workflow:



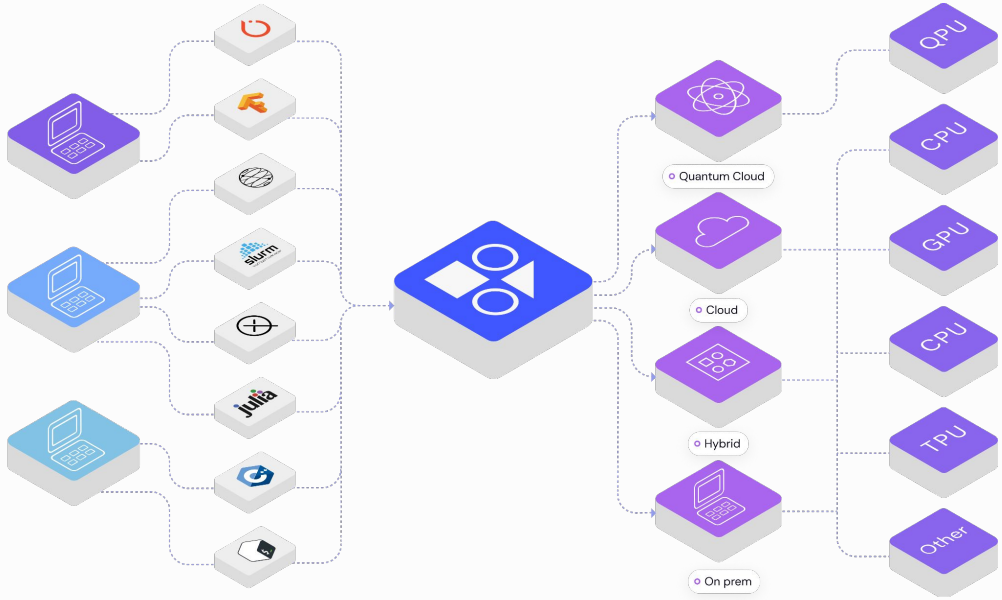


Covalent.

Covalent is an open source, free workflow orchestration platform for heterogeneous computing.

Covalent.

- Support for quantum + HPC
- Distributed computation
- Heterogeneous hardware and software
- Workflow management interface



AgnostiqHQ/covalent

Why does Covalent exist?

Computational research

Time spent on research

Time spent on Writing, Debugging, Executing, Maintaining research code

How Covalent can help.

Solution 1.

Hybrid research / experiments. A single QML experiment now contains CPU + GPU/TPU + QPU.

Solution 2.

HPC research requires rapid prototyping and experimentation across software parameters.

Solution 3.

Execute high-throughput calculations. Run massively parallel jobs at scale across clusters.

Solution 4.

Avoid long HPC queue times and redeploy to different quantum queues using serverless HPC.

Where Covalent fits in the (Python) stack.

Covalent supports a growing ecosystem of hardware and software.

Classical Resources

- Slurm executor
- AWS Fargate executor
- AWS Batch executor
- Azure executor
- GCP executor
- Kubernetes executor

Quantum Resources

- AWS-Braket Jobs
- Qiskit runtime
- More to come
- **Write your own plugin!**

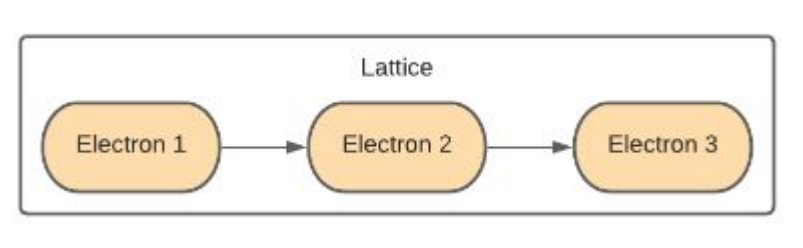
Software Packages

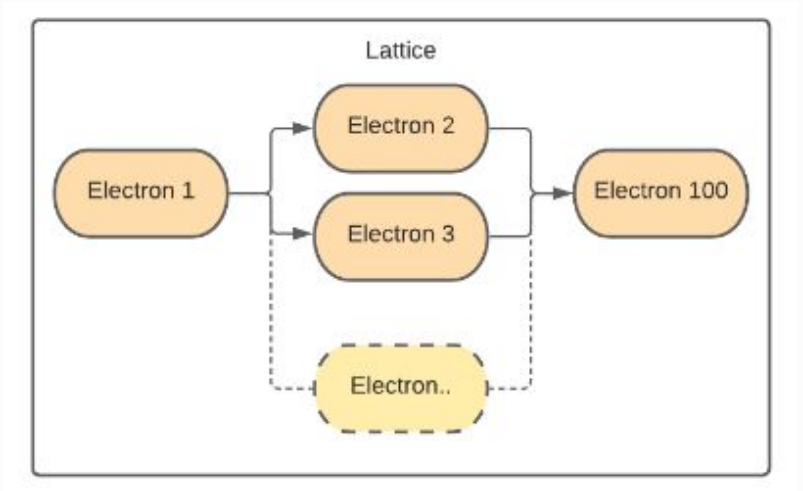
Any and all packages!

Languages

Covalent Concepts.

- Workflows are comprised of inter-dependent tasks
- Terminology adopted from Condensed Matter
- Workflows are called “lattices”
- Tasks are called “electrons” – these are fundamental blocks
- Tasks can be written in other languages, called from Python
- Tasks can each run on different hardware backends
- Python objects – not just files – are passed between tasks
- Workflows help us visually communicate experiments



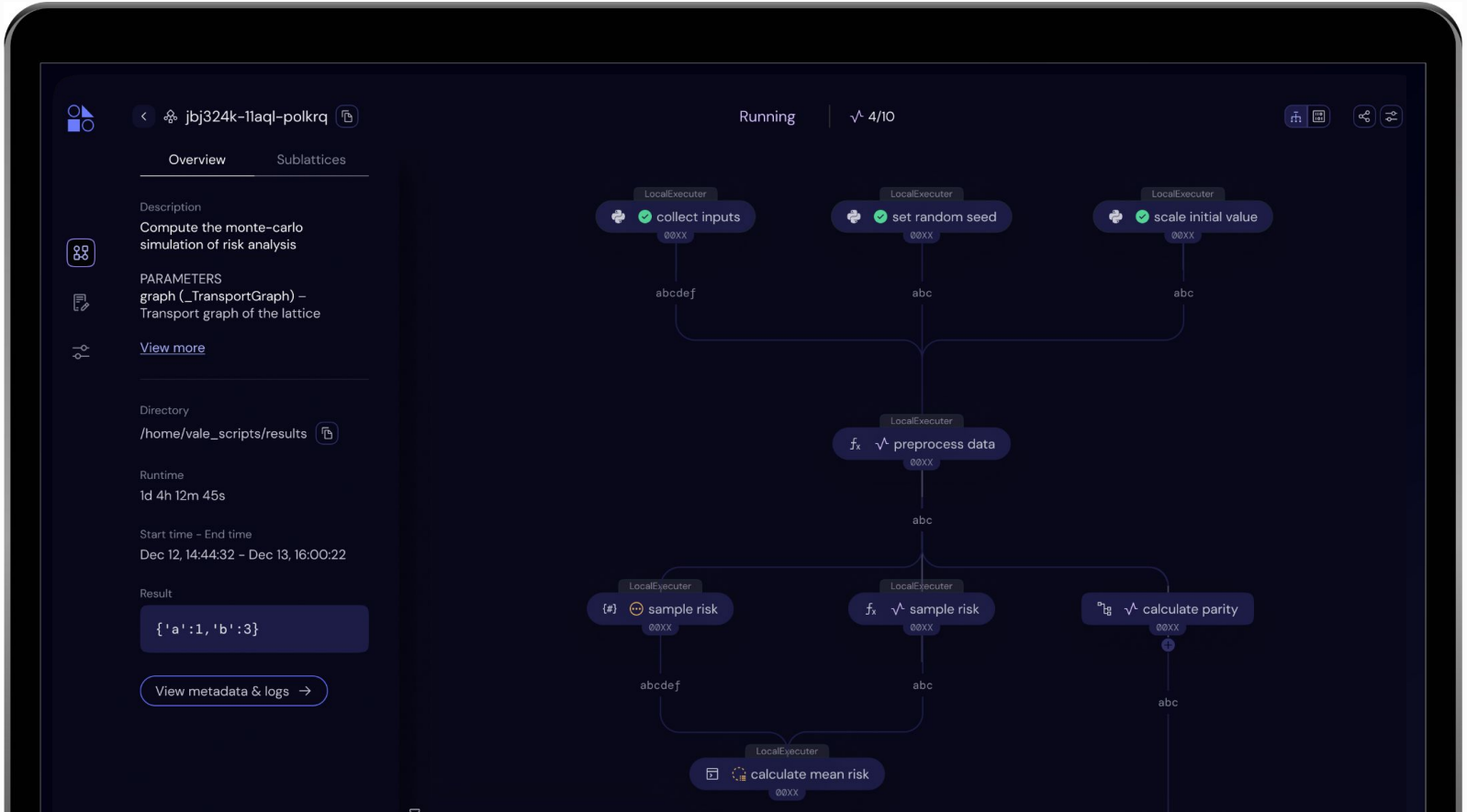


Minimal code changes required.

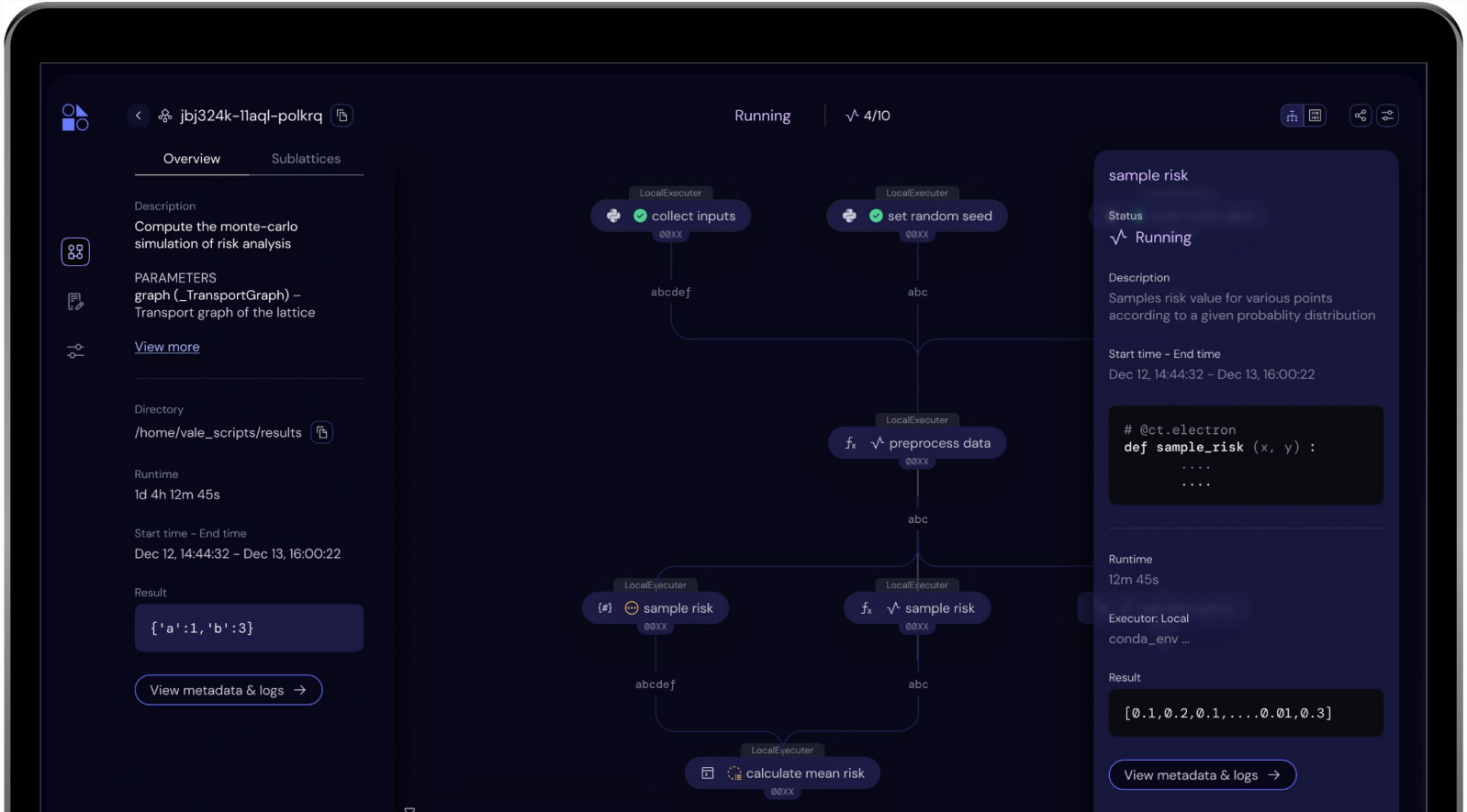
```python
def task(x, y):
    z = optimize(x, y)
    return z

def train(model):
    import tensorflow as tf
    tf.train(model)
    return model

def workflow(inputs):
    model = task(inputs[0], inputs[1])
    return train(model)

print(workflow([1,2]))
```

```python
import covalent as ct

@ct.electron(executor=AWSBatchExecutor())
def task(x, y):
    z = optimize(x, y)
    return z

@ct.electron(executor=SlurmExecutor())
def train(model):
    import tensorflow as tf
    tf.train(model)
    return model

@ct.lattice
def workflow(inputs):
    model = task(inputs[0], inputs[1])
    return train(model)

dispatch_id = ct.dispatch(workflow)([1,2])
print(ct.get_result(dispatch_id))
```

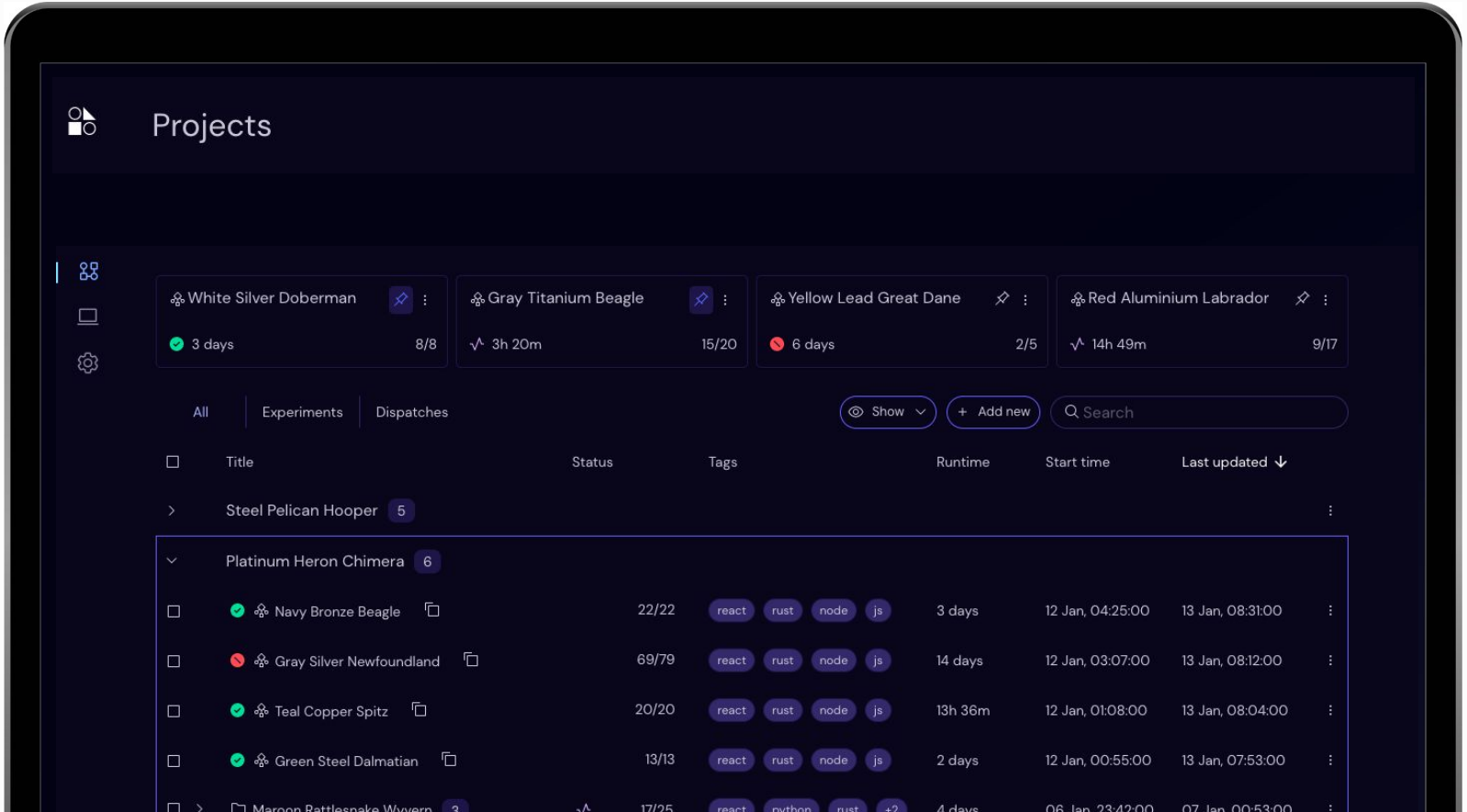
Visualize and monitor complex workflows.



Inspect task-level results and metadata.



Organize jobs across projects and experiments.



You can contribute too!

Pythonic workflows	Automatic checkpointing	Multiple language support	Little-to-no overhead	Customizable
Reproducibility	Code locally, run anywhere		Intuitive User-interface	Natively hybrid workflows
Native parallelization				Variety of executors
Code isolation	Parameter caching	Cloud agnostic	Interactive jobs	Start locally and scale

Code Examples

Curve Fitting

1. Functionalize your code

Unstructured

```
import numpy as np
import matplotlib.pyplot as plt
x = [1,2,3,9], y = [1,4,1,3]

# Fit
fit = np.polyfit(x, y, 3)
xnew = np.linspace(x[0], x[-1], 50)
y_new = fit(xnew)

# plot
fig, ax = plt.subplots()
plt.plot(x, y, 'o', xnew, y_new)
plt.xlim([x[0]-1, x[-1] + 1])
```

Structured

```
import numpy as np
import matplotlib.pyplot as plt

def fit_xy(x,y):
    z = np.polyfit(x, y, 3)
    return np.poly1d(z)

def plot_fit(x,y,xnew,fit):
    y_new=fit(xnew)
    fig,ax=plt.subplots()
    plt.plot(x,y,'o', x_new, y_new)
    plt.xlim([x[0]-1, x[-1] + 1 ])
    return fig

def exp(x,y):
    x_new = np.linspace(x[0], x[-1], 50)
    fit=fit_xy(x=x,y=y)
    return plot_fit(x=x,y=y,xnew=x_new,fit=fit)

exp(x=[1,2,3,9],y=[1,4,1,3])
```

2. Decorate your functions

```python
def fit_xy(x,y):
    z = np.polyfit(x, y, 3)
    return np.poly1d(z)
```



```python
import covalent as ct

@ct.electron
def fit_xy(x,y):
    z = np.polyfit(x, y, 3)
    return np.poly1d(z)
```

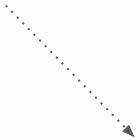
```python
def plot_fit(x,y,xnew,fit):
    y_new=fit(xnew)
    fig,ax=plt.subplots()
    plt.plot(x,y,'o', x_new, y_new)
    plt.xlim([x[0]-1, x[-1] + 1 ])
    return fig
```



```python
@ct.electron
def plot_fit(x,y,xnew,fit):
    y_new=fit(xnew)
    fig,ax=plt.subplots()
    plt.plot(x,y,'o', x_new, y_new)
    plt.xlim([x[0]-1, x[-1] + 1 ])
    return fig
```

3. Create your workflow

```
def exp(x, y):
  x_new = np.linspace(x[0], x[-1], 50)
  fit = fit_xy(x, y)
  return plot_fit(x, y, x_new, fit)
```



```
@ct.lattice
def exp(x, y):
  x_new = np.linspace(x[0], x[-1], 50)
  fit = fit_xy(x=x, y=y)
  return plot_fit(x=x, y=y, xnew=x_new, fit=fit)
```

4. Dispatch your workflow

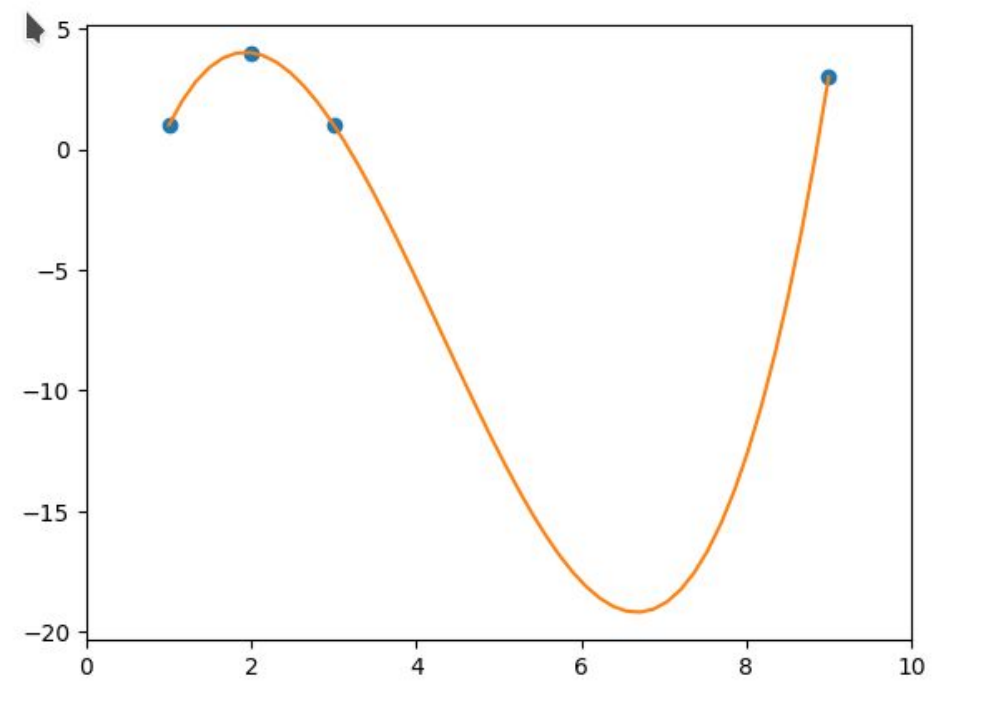
```
dispatch_id = ct.dispatch(workflow)(x=[1,2,3,9], y = [1,4,1,3])
```

5. Get the result

```
result = ct.get_result(dispatch_id, wait=True)
```

5. View results on UI

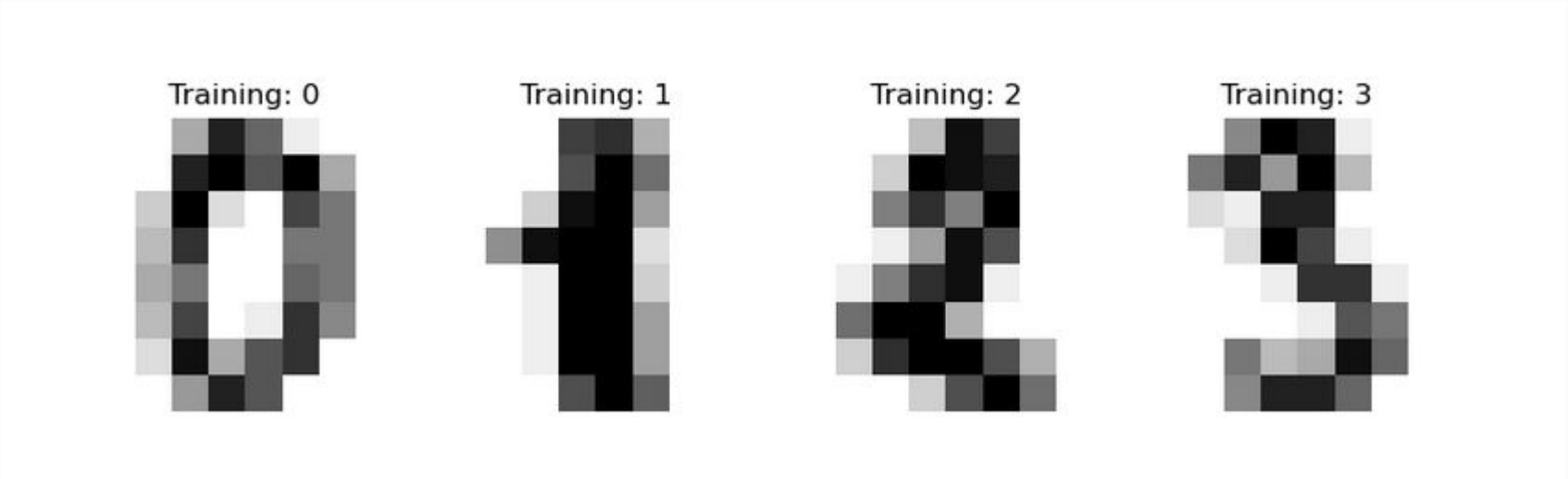
Curve fitted plot



Classification
using
Support Vector Machines

Recognizing handwritten digits

- Example adopted from [scikit-learn] tutorials
- Train a SVM classifier to recognize handwritten digits from 0-9
- Dataset consists of 8 x 8 pixel images of digits



1. Load the dataset

```
import covalent as ct
from sklearn import datasets, svm, metrics
from sklearn.model_selection import
train_test_split

@ct.electron
def load_dataset():
  return datasets.load_digits()
```

```
print(load_dataset().data)
array([[ 0.,  0.,  5., ...,  0.,  0.,  0.],
       [ 0.,  0.,  0., ..., 10.,  0.,  0.],
       [ 0.,  0.,  0., ..., 16.,  9.,  0.],
       ...,
       [ 0.,  0.,  1., ...,  6.,  0.,  0.],
       [ 0.,  0.,  2., ..., 12.,  0.,  0.],
       [ 0.,  0., 10., ..., 12.,  1.,  0.]])
```

```
print(load_dataset().target_names)
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
print(load_dataset().feature_names)
array(['pixel_0_0', 'pixel_0_1', 'pixel_0_2', 'pixel_0_3', 'pixel_0_4',
       'pixel_0_5', 'pixel_0_6', 'pixel_0_7', 'pixel_1_0', 'pixel_1_1',
       'pixel_1_2', 'pixel_1_3', 'pixel_1_4', 'pixel_1_5', 'pixel_1_6',
       'pixel_1_7', 'pixel_2_0', 'pixel_2_1', 'pixel_2_2', 'pixel_2_3',
       'pixel_2_4', 'pixel_2_5', 'pixel_2_6', 'pixel_2_7', 'pixel_3_0',
       'pixel_3_1', 'pixel_3_2', 'pixel_3_3', 'pixel_3_4', 'pixel_3_5',
       'pixel_3_6', 'pixel_3_7', 'pixel_4_0', 'pixel_4_1', 'pixel_4_2',
       'pixel_4_3', 'pixel_4_4', 'pixel_4_5', 'pixel_4_6', 'pixel_4_7',
       'pixel_5_0', 'pixel_5_1', 'pixel_5_2', 'pixel_5_3', 'pixel_5_4',
       'pixel_5_5', 'pixel_5_6', 'pixel_5_7', 'pixel_6_0', 'pixel_6_1',
       'pixel_6_2', 'pixel_6_3', 'pixel_6_4', 'pixel_6_5', 'pixel_6_6',
       'pixel_6_7', 'pixel_7_0', 'pixel_7_1', 'pixel_7_2', 'pixel_7_3',
       'pixel_7_4', 'pixel_7_5', 'pixel_7_6', 'pixel_7_7'], dtype='<U9')
```

2. Build the SVM classifier

```
@ct.electron
def build_classifier(gamma: float):
  return svm.SVC(gamma = gamma)
```

3. Split the dataset into training/test sets

```
@ct.electron
def split_data(features, targets, test_set_size):
  x_train, x_test, y_train, y_test = train_test_split(features, targets,
          test_size=test_set_size, shuffle=False)
  return x_train, x_test, y_train, y_test
```

4. Train the classifier

```
@ct.electron
def train_classifier(clf, features, targets):
  return clf.fit(features, targets)
```

5. Get model predictions

```
@ct.electron
def get_predictions(clf, test_features):
  return clf.predict(test_features)
```

6. Generate classification report

```
@ct.electron
def get_classification_report(y_test, predictions):
  return metrics.classification_report(y_test, predictions)
```

7. Create workflow

```
@ct.lattice
def workflow(gamma: float):
  # Load data
  dataset = load_dataset()
  # Build classifier
  clf = build_classifier(gamma=gamma)
  # split data
  x_train, x_test, y_train, y_test = split_data(features=dataset.data,
                                                targets=dataset.target,
                                                test_set_size=0.5)
  # Train classifier
  clf = train_classifier(clf, features=x_train, targets=y_train)
  # Predictions
  predictions = get_predictions(clf, x_test)

  return predictions
```

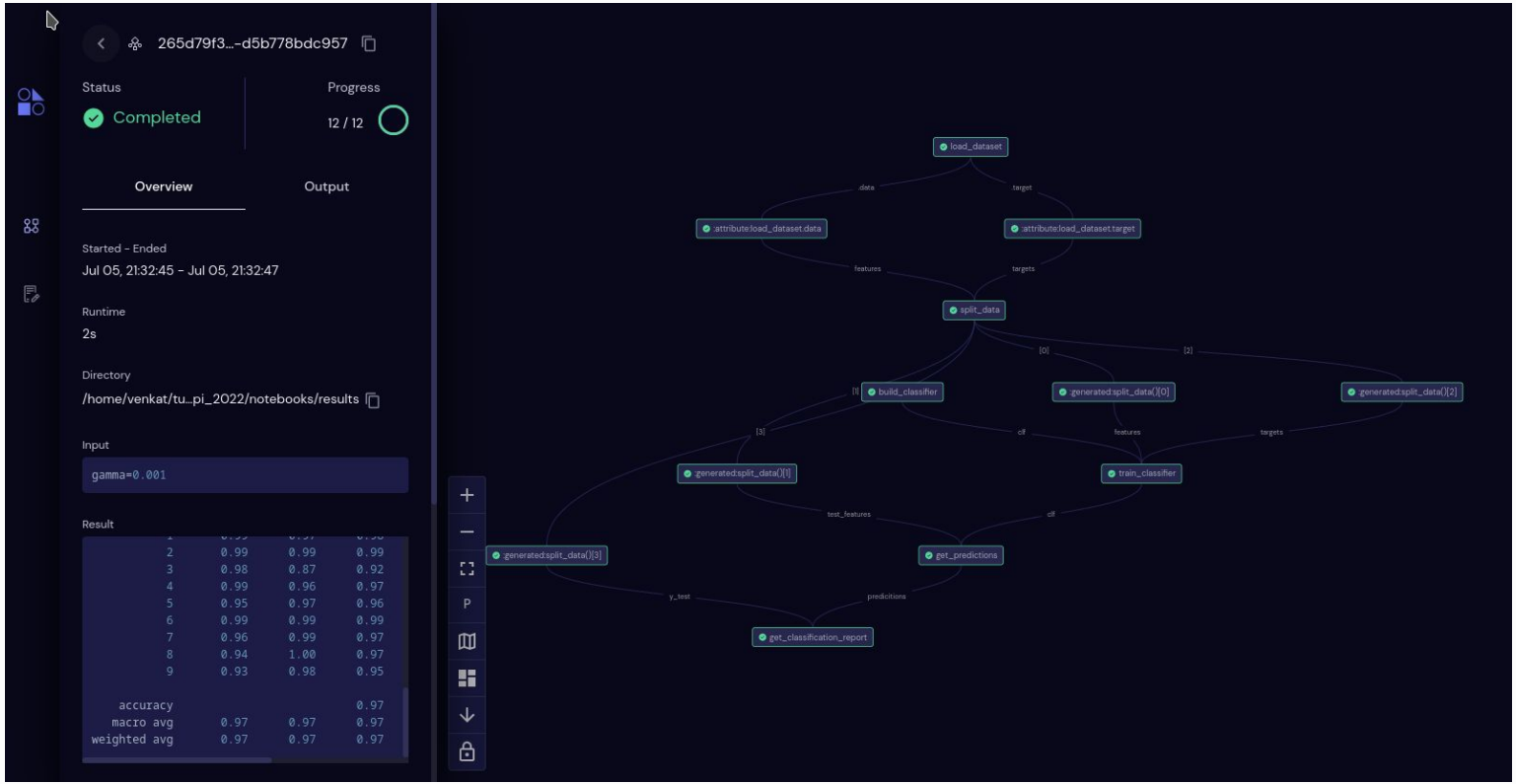
Dispatch

```
dispatch_id = ct.dispatch(workflow)(0.001)
```

Get result

```
result = ct.get_result(dispatch_id, wait=True)
```

Workflow execution



Classification report

	precision	recall	f1-score	support
0	1.00	0.99	0.99	88
1	0.99	0.97	0.98	91
2	0.99	0.99	0.99	86
3	0.98	0.87	0.92	91
4	0.99	0.96	0.97	92
5	0.95	0.97	0.96	91
6	0.99	0.99	0.99	91
7	0.96	0.99	0.97	89
8	0.94	1.00	0.97	88
9	0.93	0.98	0.95	92
accuracy			0.97	899
macro avg	0.97	0.97	0.97	899
weighted avg	0.97	0.97	0.97	899

Sublattices

- We have a workflow that trains the classifier for a given value of **gamma**
- The value was chosen arbitrarily
- We can do hyper-parameter tuning on the model to find the effects of changing **gamma** values
- To this end, Covalent allows one to build larger workflows by building **sublattices**

```
@ct.electron
@ct.lattice
def workflow(gamma: float):
  # Load data
  dataset = load_dataset()
  # Build classifier
  clf = build_classifier(gamma=gamma)
  # split data
  x_train, x_test, y_train, y_test = split_data(features=dataset.data,
                                                targets=dataset.target,
                                                test_set_size=0.5)
  # Train classifier
  clf = train_classifier(clf, features=x_train, targets=y_train)
  # Predictions
  predictions = get_predictions(clf, x_test)

  return predictions
```

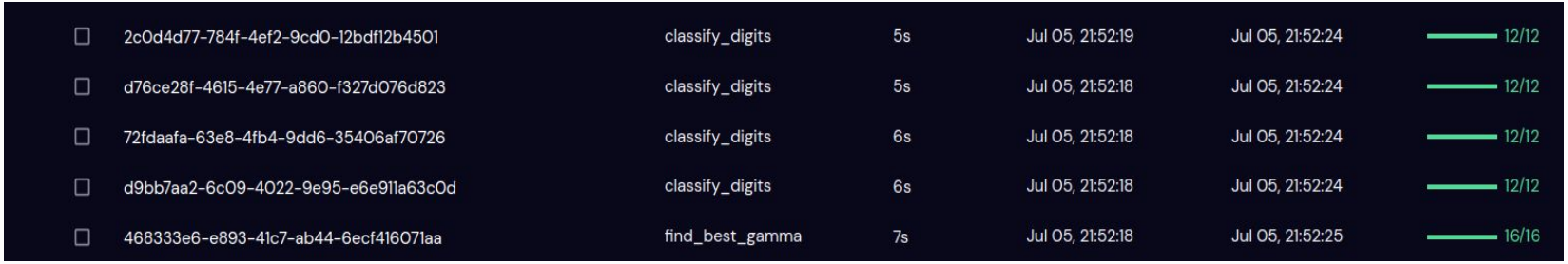
Larger workflow

```python
@ct.lattice
def find_best_gamma(gamma_values):
    results = {}
    for gamma in gamma_values:
        results[f"{gamma}"] = {}
        predictions, clf_report, _ = classify_digits(gamma)

        results[f"{gamma}"]["predictions"] = predictions
        results[f"{gamma}"]["report"] = clf_report
    return results
```

- We invoke the previous workflow as if it were just an electron (**sublattice**)
- We can now dispatch this for different **gamma** values and collect the results

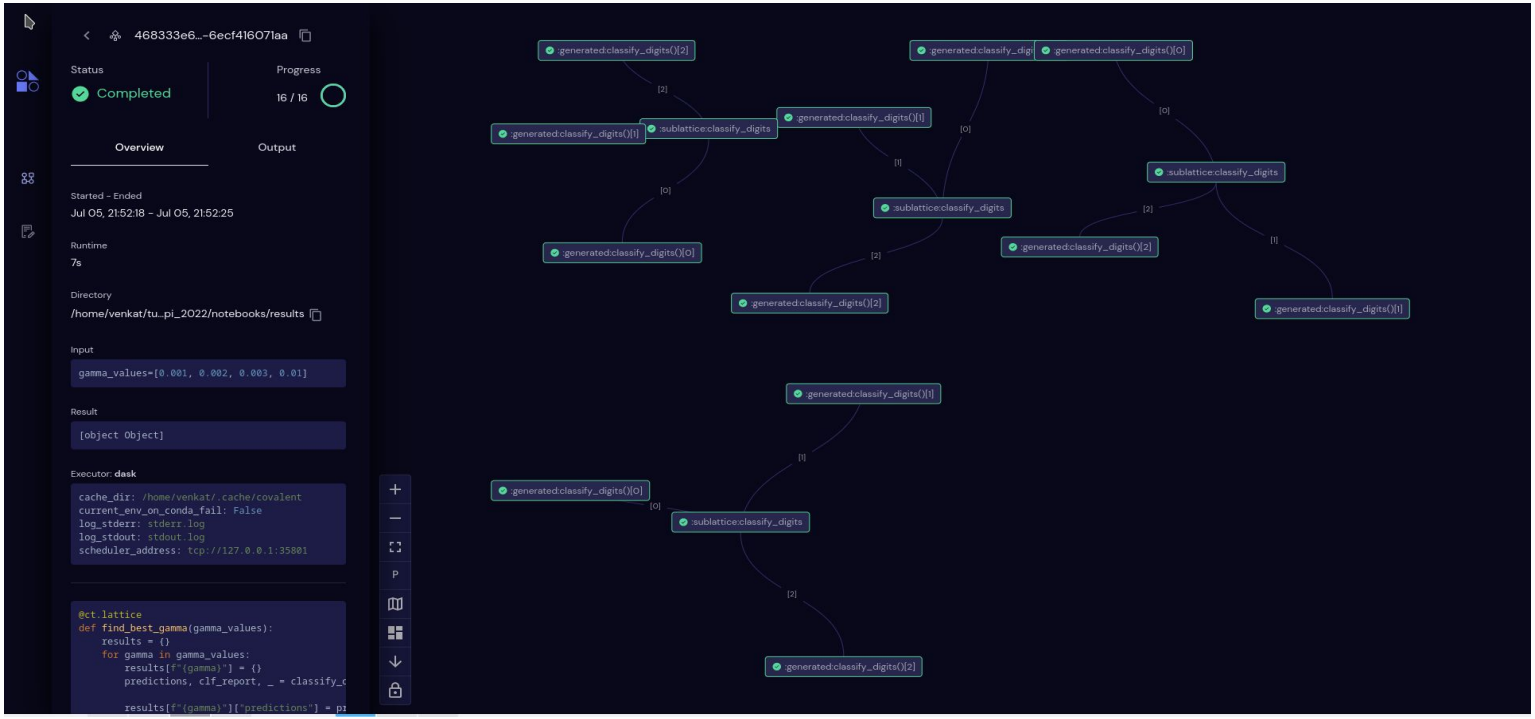
```python
result = ct.dispatch_sync(find_best_gamma)([0.001, 0.002, 0.003, 0.01])
```

Sublattice submission



☐	2c0d4d77-784f-4ef2-9cd0-12bdf12b4501	classify_digits	5s	Jul 05, 21:52:19	Jul 05, 21:52:24	12/12
☐	d76ce28f-4615-4e77-a860-f327d076d823	classify_digits	5s	Jul 05, 21:52:18	Jul 05, 21:52:24	12/12
☐	72fdaafa-63e8-4fb4-9dd6-35406af70726	classify_digits	6s	Jul 05, 21:52:18	Jul 05, 21:52:24	12/12
☐	d9bb7aa2-6c09-4022-9e95-e6e911a63c0d	classify_digits	6s	Jul 05, 21:52:18	Jul 05, 21:52:24	12/12
☐	468333e6-e893-41c7-ab44-6ecf416071aa	find_best_gamma	7s	Jul 05, 21:52:18	Jul 05, 21:52:25	16/16

Graph with Sublattices



Bohemian Eigenvalues

http://www.bohemianmatrices.com

What are they?

- Stands for bounded height integer matrix eigenvalues
- A family of Bohemian matrices is a distribution of random matrices where the matrix entries are sampled from a discrete set of bounded height integers
- The discrete set must be independent of the dimension of the matrices.
- **Bohemian Eigenvalues**
  - Eigenvalues of a family of Bohemian matrices.
- In our example we will consider 5 x 5 random matrices generated from the set [−1, 0, 1]

Bohemian Matrices

```
import numpy as np

np.random.choice([-1, 0, 1], 25).reshape(5, 5)

array([[ 0,  1,  0,  0,  1],
       [ 1,  0,  1,  0, -1],
       [ 0,  0,  0, -1, -1],
       [ 1, -1,  0,  1, -1],
       [-1,  1, -1, -1,  0]])
```

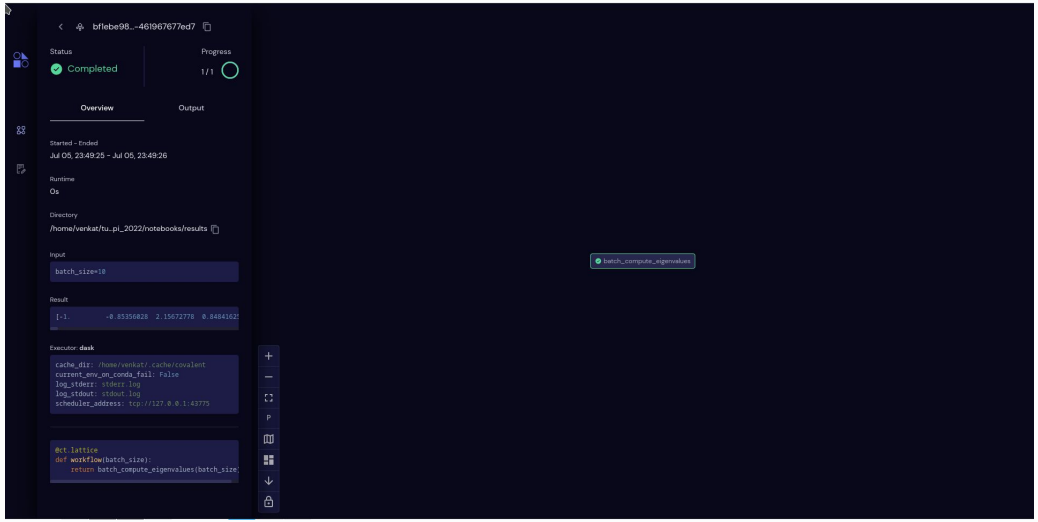
Lets compute the eigenvalues of a lot of such matrices and view the results

Workflow Electrons

```
@ct.electron
def generate_random_matrix(integer_set, nrows, ncols):
    return np.random.choice(integer_set, nrows*ncols).reshape(nrows, ncols)


@ct.electron
def compute_eigenvalues(matrix):
    eig, _ = np.linalg.eig(matrix)
    return eig)


@ct.electron
def get_real_part(eigenvalues):
    return [np.real(eig) for eig in eigenvalues]

@ct.electron
def get_imag_part(eigenvalues):
    return [np.imag(eig) for eig in eigenvalues]

@ct.electron
def batch_compute_eigenvalues(batch_size):
    res = []
    matrices = [generate_random_matrix([-1, 0, 1], 5, 5) for i in range(batch_size)]
    for matrix in matrices:
        res.append(compute_eigenvalues(matrix))
    return (get_real_part(res), get_imag_part(res),)
```

Single node workflow

```
@ct.lattice
def workflow(batch_size):
    return batch_compute_eigenvalues(batch_size)
```



Remote executors

- With Covalent we can choose to execute certain tasks of a workflow on different hardware/machines of our choice
- This is made possible through the use of **executors**
- Executors allow the user to customize the execution environment of a particular task in the workflow
- Executors are plugins to Covalent and can be installed separately via **pip**
- Using executors, users can dispatch a task to a remote machine or a HPC cluster if access is available
- Supported executors
    - Local, Dask, SSH, Slurm/LSF/PBS, Kubernetes, AWS Fargate, AWS Batch, Qiskit Runtime

Remote executors

- Since computing eigenvalues of matrices (large matrices) is potentially compute heavy
- Offload the computation to a cloud HPC slurm cluster using Covalent's Slurm executor
- To do so, we simply pass an instance of the executor to the lattice decorator for the workflow

Remote Executor: SLURM

```
from covalent.executor import SlurmExecutor

slurm = SlurmExecutor(
    username="test",
    address="<server address>",
    ssh_key_file="~/.ssh/my_key",
    conda_env = "env_name",
    poll_freq=60,
    remote_workdir="/home/test",
    options={
        "partition": "<partition>",
        "cpus-per-task": <cpus>,
        "ntasks": <ntasks>
    }
)
```

```
@ct.lattice(executor=slurm)
def workflow(batch_size):
    return batch_compute_eigenvalues(batch_size)
```

Beehive Cluster

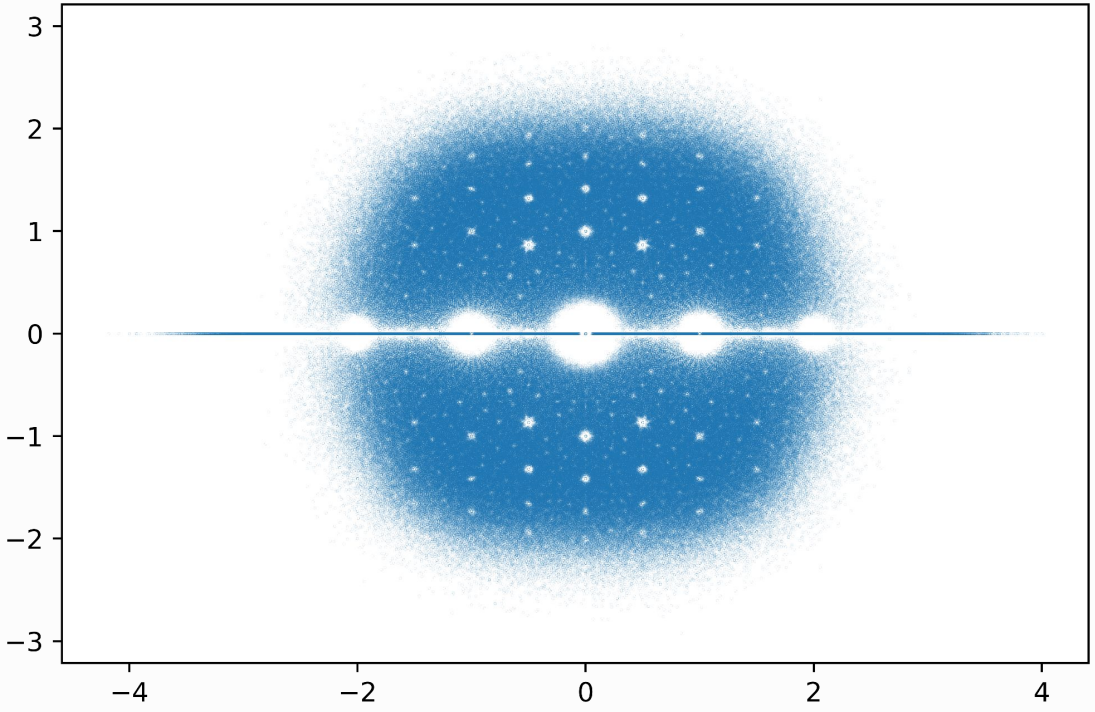
Beehive cluster

```
(base) venkat@main:~$ sinfo
       CLUSTER                    HOSTNAMES   PARTITION      STATE   CPUS     MEMORY              GRES
           N/A rpi-main-compute-dy-c44xlarg rpi-main-co      idle~     16          1             (null)
           N/A rpi-main-compute-dy-c44xlarg rpi-main-co      idle~     16          1             (null)
           N/A rpi-main-compute-dy-c44xlarg rpi-main-co      idle~     16          1             (null)
           N/A rpi-main-compute-dy-c44xlarg rpi-main-co      idle~     16          1             (null)
           N/A rpi-main-compute-st-c44xlarg rpi-main-co       idle     16          1             (null)
           N/A rpi-main-compute-st-c44xlarg rpi-main-co       idle     16          1             (null)
           N/A rpi-main-compute-st-c44xlarg rpi-main-co       idle     16          1             (null)
           N/A rpi-main-compute-st-c44xlarg rpi-main-co       idle     16          1             (null)
           N/A rpi-main-debug-dy-t22xlarge- rpi-main-de      idle~      8          1             (null)
           N/A rpi-main-debug-dy-t22xlarge- rpi-main-de      idle~      8          1             (null)
           N/A rpi-main-debug-dy-t22xlarge- rpi-main-de      idle~      8          1             (null)
           N/A rpi-main-debug-dy-t22xlarge- rpi-main-de      idle~      8          1             (null)
           N/A rpi-main-debug-st-t22xlarge- rpi-main-de       idle      8          1             (null)
           N/A rpi-main-debug-st-t22xlarge- rpi-main-de       idle      8          1             (null)
           N/A rpi-main-debug-st-t22xlarge- rpi-main-de       idle      8          1             (null)
           N/A rpi-main-debug-st-t22xlarge- rpi-main-de       idle      8          1             (null)
(base) venkat@main:~$
```

Job queue after dispatch

```
Every 0.1s: squeue                                                                  main.rpi: Wed Jul  6 05:27:51 2022

             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
                34 rpi-main- slurm-b6   venkat  R       0:06      1 rpi-main-debug-st-t22xlarge-1
```

Result

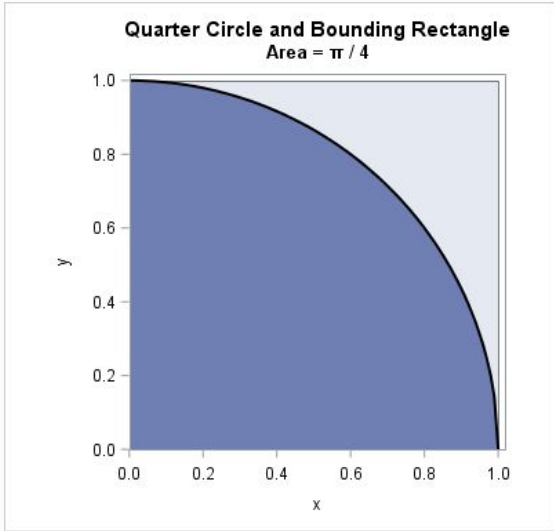


Interfacing with other languages

- Covalent is designed to be **language agnostic**
- If there are bindings between Python and the language of choice, workflows can be constructed where electrons invoke code written in different languages
- Here we demonstrate how one can interface with a module written in C through Covalent
- More language support coming
  - Bash
  - Julia
  - C++
  - Fortran
  - (Request your favorite language on GitHub)

Monte Carlo estimate of PI

- Using area under the curve of a quarter circle



Python/C API

```
#include <Python.h>

static PyObject* compute_pi(PyObject* self, PyObject* args) {
    unsigned int partitions;
    double area = 0.0;
    double dh;

    if (!PyArg_ParseTuple(args, "I", &partitions)) {
        return NULL;
    }
    dh = 1.0/partitions;

    for (int i = 0; i < partitions; i++) {
        double x = dh * (i - 0.5);
        area += (4.0/(1.0 + x*x));
    }
    // pi approximation
    return PyFloat_FromDouble(area*dh);
}
```

Create Python module (pimonte)

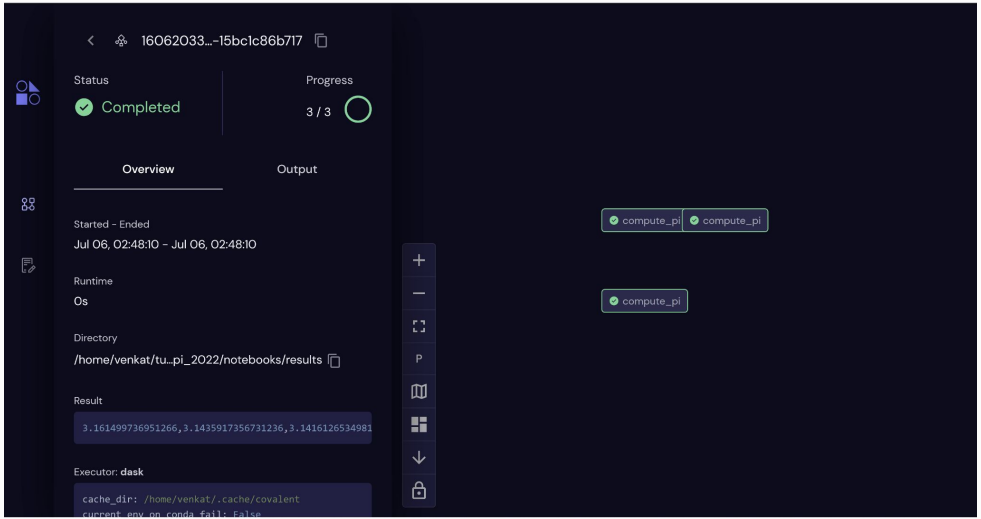
```
static PyMethodDef PimonteMethods[] = {
    {"compute_pi",  compute_pi, METH_VARARGS, "Compute an approximation to PI"},
    {NULL, NULL, 0, NULL}        /* Sentinel */
};

static struct PyModuleDef pimonte = {
    PyModuleDef_HEAD_INIT,
    "pimonte",   /* name of module */
    NULL, /* module documentation, may be NULL */
    -1,       /* size of per-interpreter state of the module,
                 or -1 if the module keeps state in global variables. */
    PimonteMethods
};

PyMODINIT_FUNC PyInit_pimonte(void) {
    return PyModule_Create(&pimonte);
}
```

Build and Install

```
from distutils.core import setup, Extension

pimonte = Extension('pimonte', sources=['pimonte.c'],
    extra_compile_args=["-O3"])

setup(name='pimonte',
    version='1.0',
    description='A approximation to PI',
    ext_modules=[pimonte])



python setup.py build
```

Covalent workflow

```
import covalent as ct
import pimonte

@ct.electron
def compute_pi(n):
    return pimonte.compute_pi(n)

@ct.lattice
def workflow():
    res = []
    for i in [100, 1000, 100000]:
        res.append(compute_pi(i))
    return res

dispatch_id = ct.dispatch(workflow)()
res = ct.get_result(dispatch_id, wait=True)

print(res.result)

[3.161499736951266, 3.1435917356731236,
3.1416126534981603]
```



High Performance Computing in the Cloud

Cloud HPC: How does it work?

Basics of Cloud HPC

- Resources are virtual templates
- Extremely efficient hypervisors carve up bare metal servers into VMs
- VMs are hierarchically grouped into *zones* (data centers) as well as *regions*
- Can be used to compare application-specific performance, costs
- Hybrid infrastructure is a big focus as orgs already have purchased infra
- Tradeoffs in costs/availability for reasons such as local power costs
- No real visibility into load on individual data centers, unlike traditional HPC

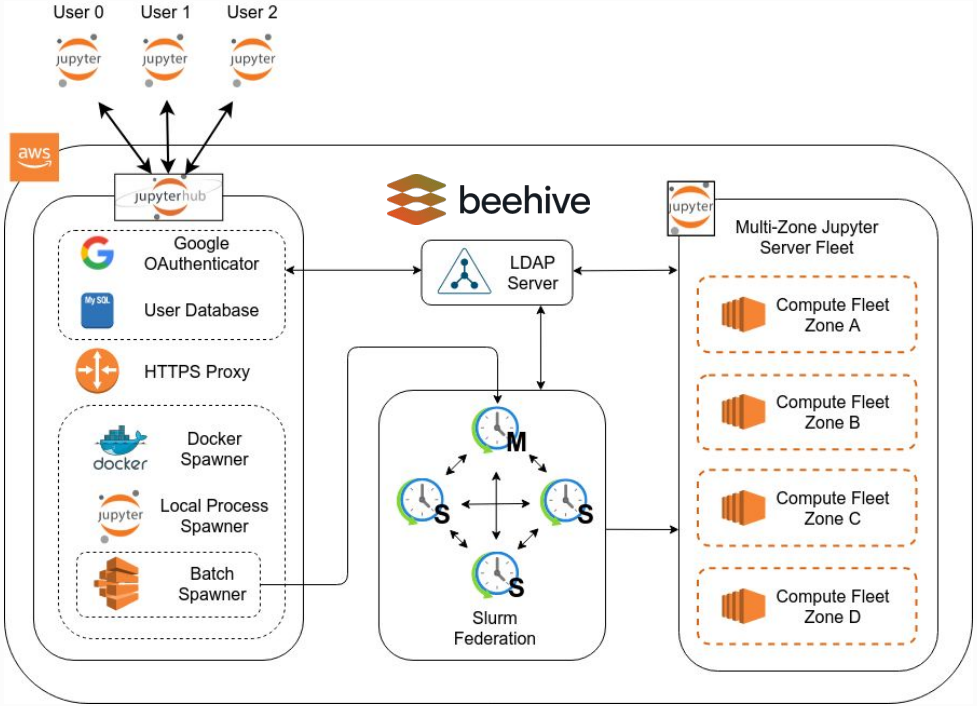
Benefits of Cloud HPC:

- On-demand, scalable resources
- Transparent pay-as-you-go pricing
- Turnkey managed platforms available
- New hardware available sooner

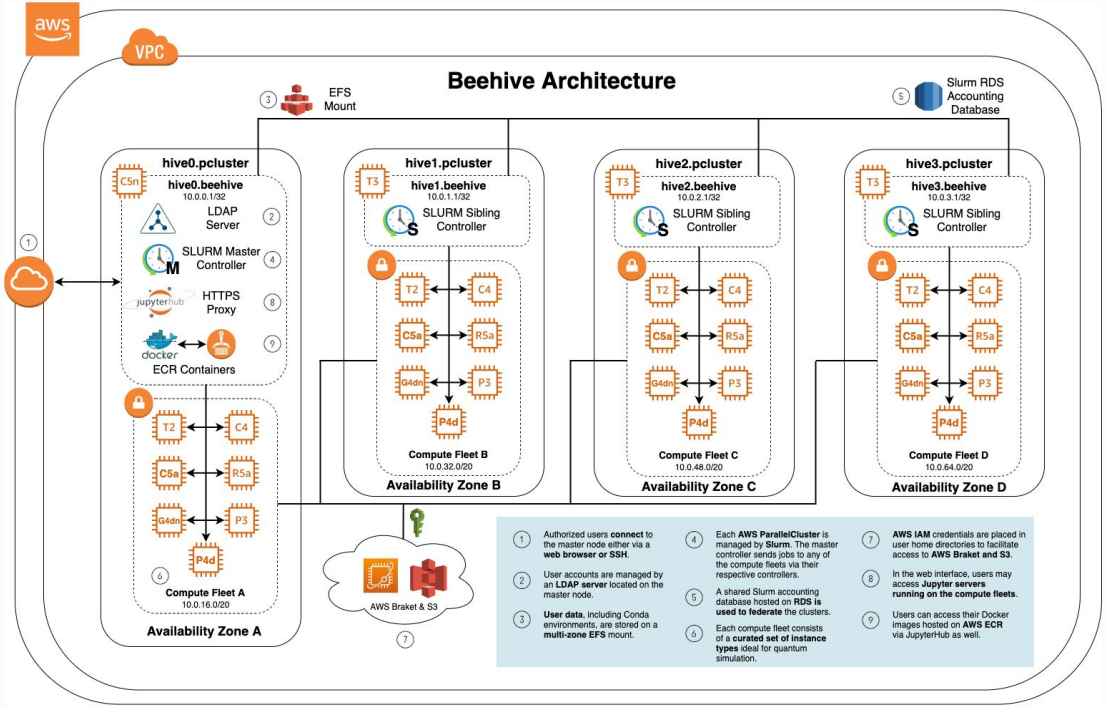
Pitfalls of Cloud HPC:

- Longer wait for resource allocation
- Proper infrastructure management is difficult and non-trivial
- Network data costs can grow quickly
- Virtualization can decrease performance

We developed Beehive for Cloud HPC prototyping



We use a tool called Beehive for HPC prototyping



- Deploy an HPC cluster with ~5 lines of code in ~20 minutes
- JupyterHub front-end, Slurm/Docker back-end, 8 real QCs available
- Ideal for tutorials with up to 100 users
- Can extend on-prem supercomputers
- Will be released open-source in the near future

How did we configure Beehive for this tutorial?

```
{
  "Name": "beehive-rpi",
  "Region": "us-east-1",
  "Hives": [{
    "ClusterName": "main",
    "AvailabilityZone": "us-east-1a",
    "MasterInstanceType": "c5n.4xlarge",
    "Partitions": [{
      "Name": "debug",
      "InstanceType": "t2.2xlarge",
      "MinCount": "4",
      "MaxCount": "8"
      },{
      "Name": "compute",
      "InstanceType": "c4.4xlarge",
      "MinCount": "8",
      "MaxCount": "16"
      },
      "Master": "true",
      "Active": "true"
  }]
}
```

```
  # Create the network infrastructure
> beehive network create -n "beehive-rpi" -r "us-east-1"
  -z "us-east-1a"

  # Create the cluster(s)
> beehive cluster create -n "beehive-rpi" -r "us-east-1"
  -k beehive-key.pem --hives hives.json

  # Federate the cluster(s)
> beehive federate -n "beehive-rpi" -r "us-east-1"
  -k beehive-key.pem --hives hives.json

  # Initialize JupyterHub
> beehive jupyter init -n "beehive-rpi" -r "us-east-1"
  --hives hives.json

  # Add a new user
> beehive user add -n "beehive-rpi" -r "us-east-1"
  -k beehive-key.pem --hives hives.json -u newuser
```

Molecular Dynamics with Covalent

Setup

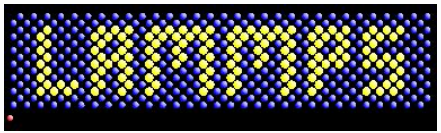
- Local
  - If you have Python < 3.8 installed...
    - Install Miniconda
    - Create a conda environment with Python v 3.8
    - **conda create -n rpi python=3.8**
    - conda activate rpi
  - pip install covalent jupyter scipy
  - **covalent start**
- Login to **rpi.tutorials.covalent.xyz** to explore Beehive
- Username
  - First two letters of first name + first 5 letters of last name
  - Password gets set upon first login
- Add a public RSA key if you want to use the Slurm executor

What is Molecular Dynamics?

- A technique for studying complex physical systems at the atomic level using computer simulations
- Equations of motion for the system are solved numerically
- Time evolution of the system is tracked
- Periodic snapshots of the system are saved for post-processing
- Several open source simulators available

AMBER MD

FAST. FLEXIBLE. FREE.
GROMACS



Molecular Dynamics

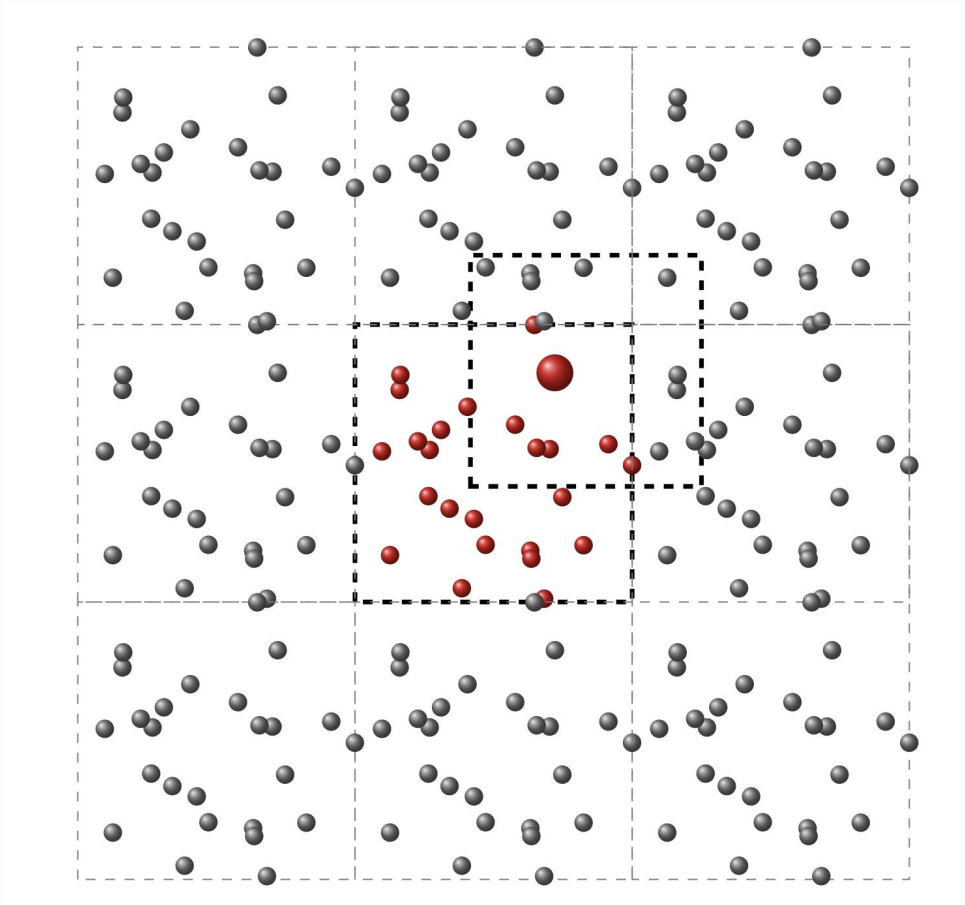
- In the atomic model, physical systems are described as collections of **classical** atoms that interact with each other by interatomic forces/potentials
- There are a plethora of well established interatomic potentials available to model various aspects of molecules and biological systems
  - CHARMM Force Fields
  - LAMMPS pair styles
- Lennard Jones potential is a widely used potential when studying soft matter systems (polymers, polymer melts, nano-composities ...)

Molecular Dynamics

- Once the interaction potentials are specified for a system the simulation domain is to be specified
- The spatial region in which the simulation is carried out
- Also defines the simulation dimension i.e. 2D/3D
- **Boundary conditions**
  - Walls
  - Moving boundaries
  - Periodic boundary conditions

Periodic Boundary conditions

- Periodic boundary conditions are widely used in MD
- Allows one to simulate smaller systems
- Typical MD system computation scale O(N)/O(N^2)
- PBC reduces the number of particles needed in the simulation by effectively generating multiple repeating copies of the system
- Particles leaving a cell enter the domain back from the opposite side (**images)**



covalent.xyz

Minimum Image Convention

- Since PBC effectively renders multiple copies of the system in each direction
- Which particles/images the interactions ought to be computed with
- **Minimum Image Convention**
- If particles i and j are more than L/2 apart, interaction of the i'th particle with jth is then effectively compute dusing *j*th nearest image

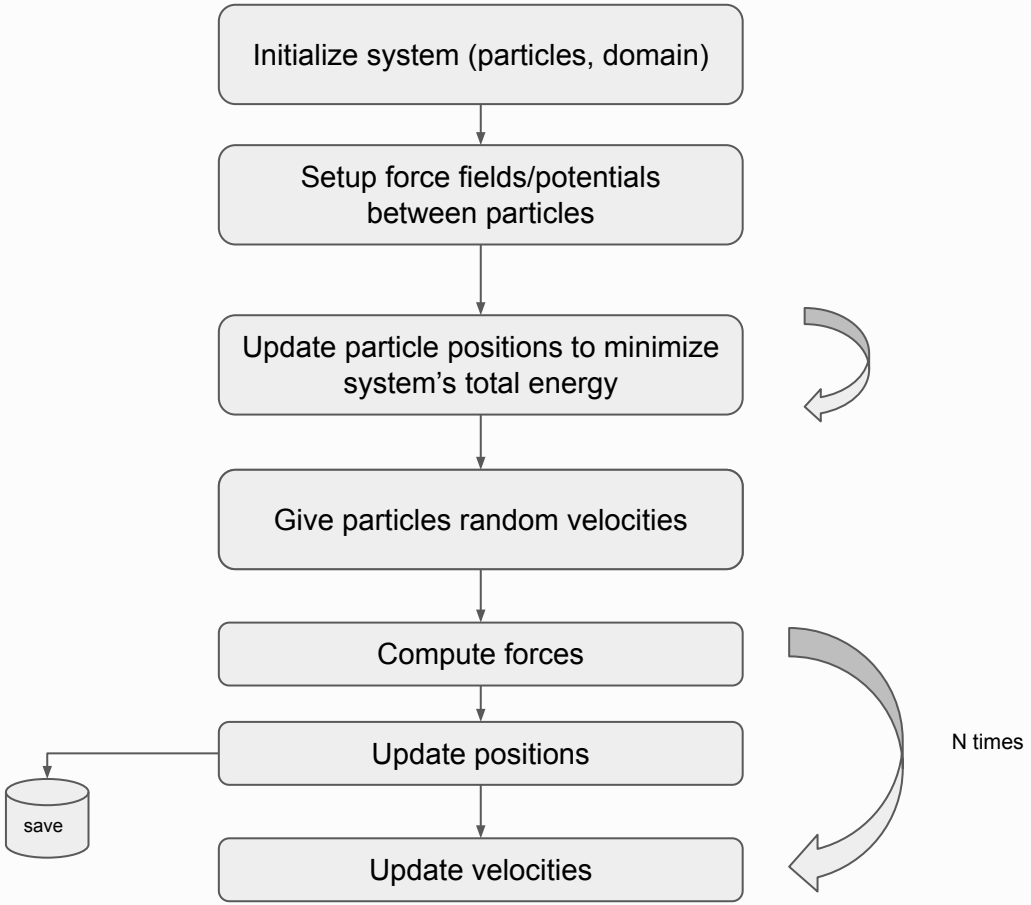
Minimization

- This is an essential part of a MD simulation
- System of particles randomly created
- Interaction potentials initialized
- Initial configuration may not be in a potential minimum
- System needs to be relaxed to a minimum energy configuration
- Gradient descent typically used
- Positions of all particles iteratively adjusted until the total PE is a minimum

Time integration

- After minimization, the system is evolved forward in time by solving the Equations of Motion
- **Velocity-Verlet** is the most popular choice as MD integrators
- Energy conserving
- Suitable for long simulations
- Efficient
- Numerically stable

MD algorithm



Tutorial

Exercises

- Run a simulation with larger number of particles
  - Hint: If minimization is taking too long, increase the system size
- Implement a **Harmonic** interaction potential

$$U(r) = \frac{1}{2}k(r - r_0)^2$$

- Dispatch workflow to Beehive using the Slurm Executor
- Study scaling of the simulation as function of number of particles

Covalent

Thank You

covalent.xyz

agnostiqHQ/covalent

@covalentxyz