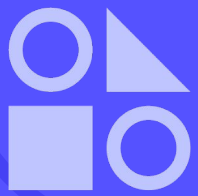
agnostiq



Covalent.

Workflow Orchestration for heterogeneous (HPC/Quantum) computing

Jack S. Baker, PhD
Quantum Algorithms Researcher

Venkat Bala, PhD
HPC Engineer
Agnostiq

Overview

1. Introduction to Covalent
2. Covalent 101
3. Code Examples
    a. Multiple dispatches, sub-lattices, results
    b. Covalent Executors
    c. C/C++ integration (Classical HPC)
4. Introduction to Quantum Machine Learning
5. Focus on Similarity Learning
6. Heterogeneous Similarity learning with Covalent **(hands-on workshop)**

Workflow Orchestration

Why is workflow orchestration is needed?

Primary Challenges:

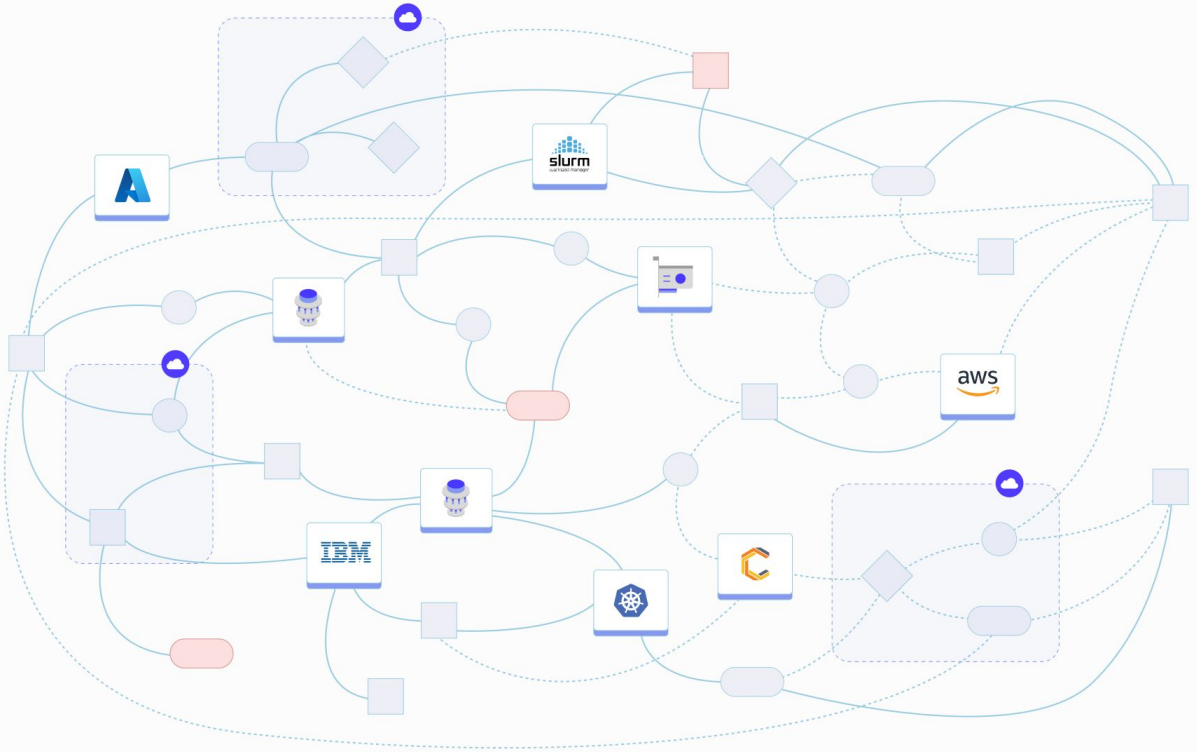
- Explore new research problems
- Apply interdisciplinary techniques
- Push the boundaries of hardware
- Find hidden patterns in data

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

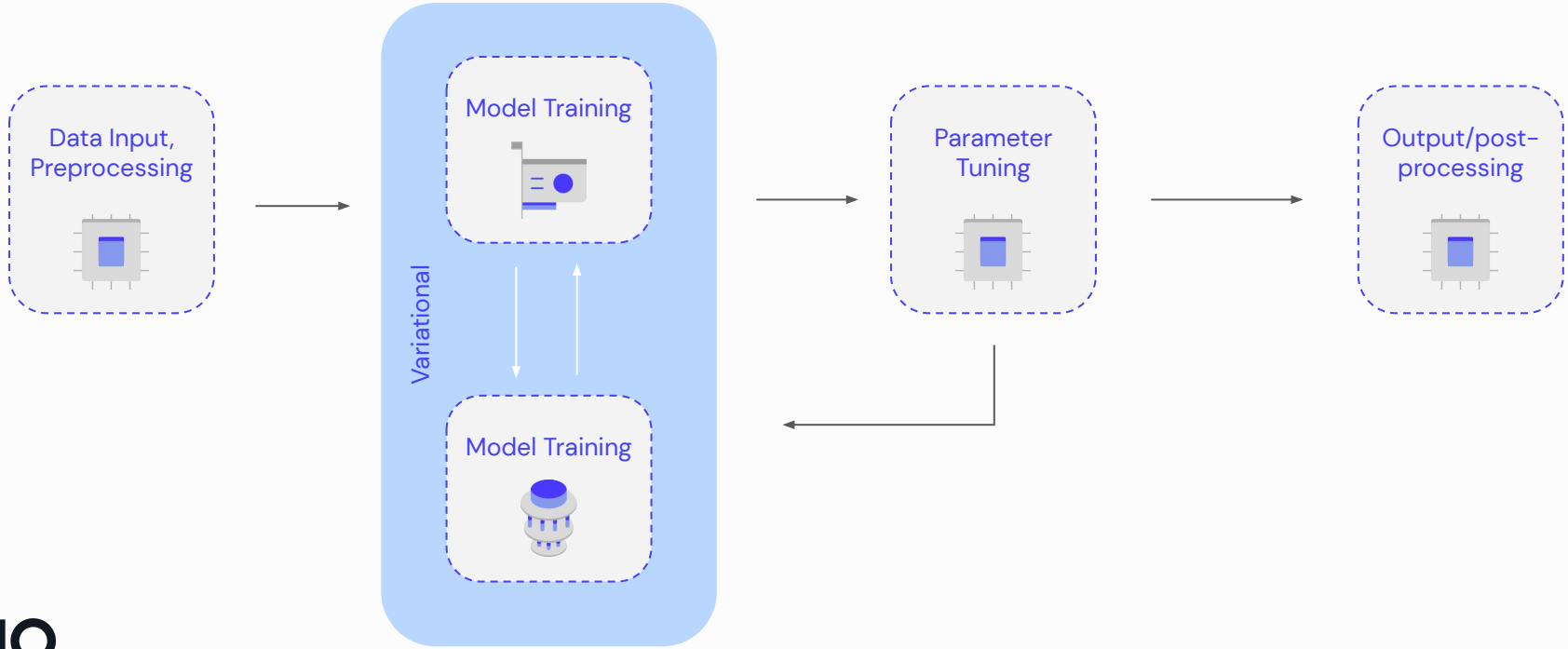
Lots of work, little reward... everyone comes up with their own bespoke solutions to these

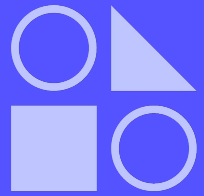
Modern computational workflows are complex.



Quantum is driving the need for more heterogeneity

A simplified quantum machine learning workflow:



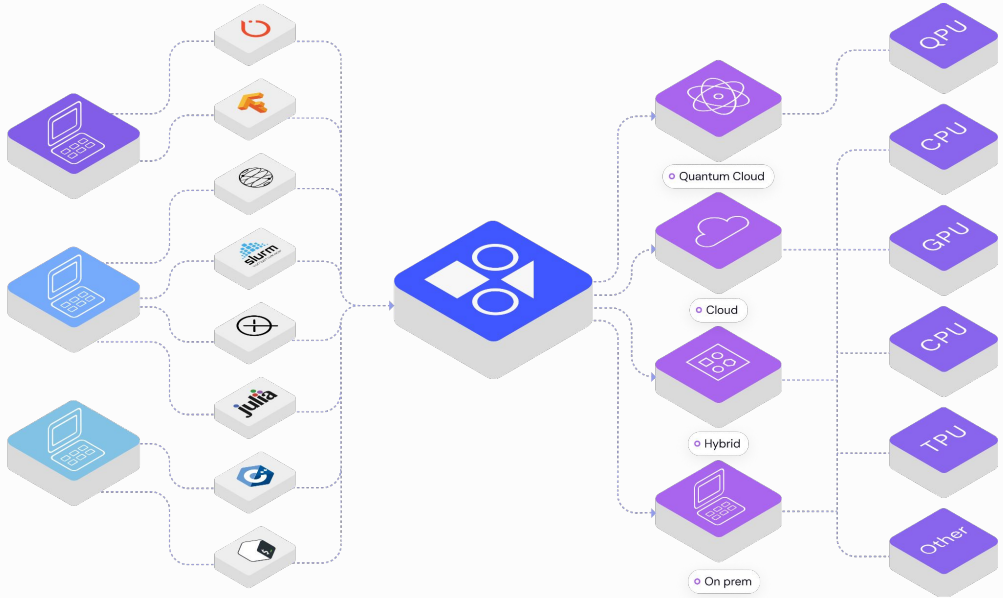


Covalent.

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 (coming soon)
- More to come
- **Write your own plugin!**

Software Packages

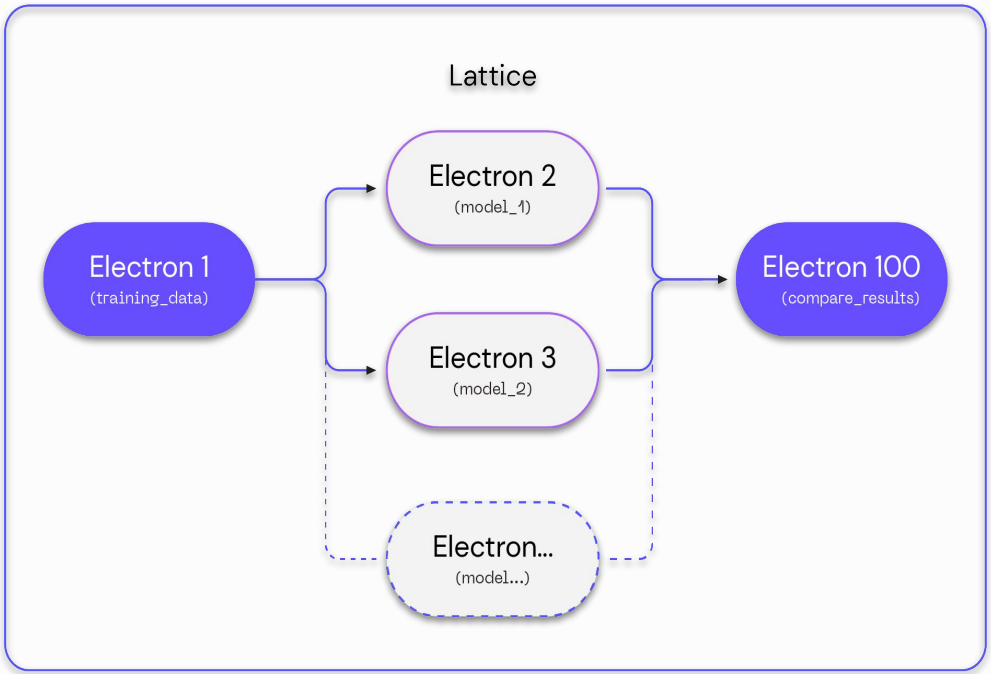
Any and all packages!

Languages

Covalent 101: The Basics

Covalent Concepts

- Workflows are comprised of inter-dependent tasks
- Workflows are called “lattices”
- Tasks are called “electrons” – these are fundamental blocks
- Tasks can be written in other languages
- Tasks can each run on different hardware backends

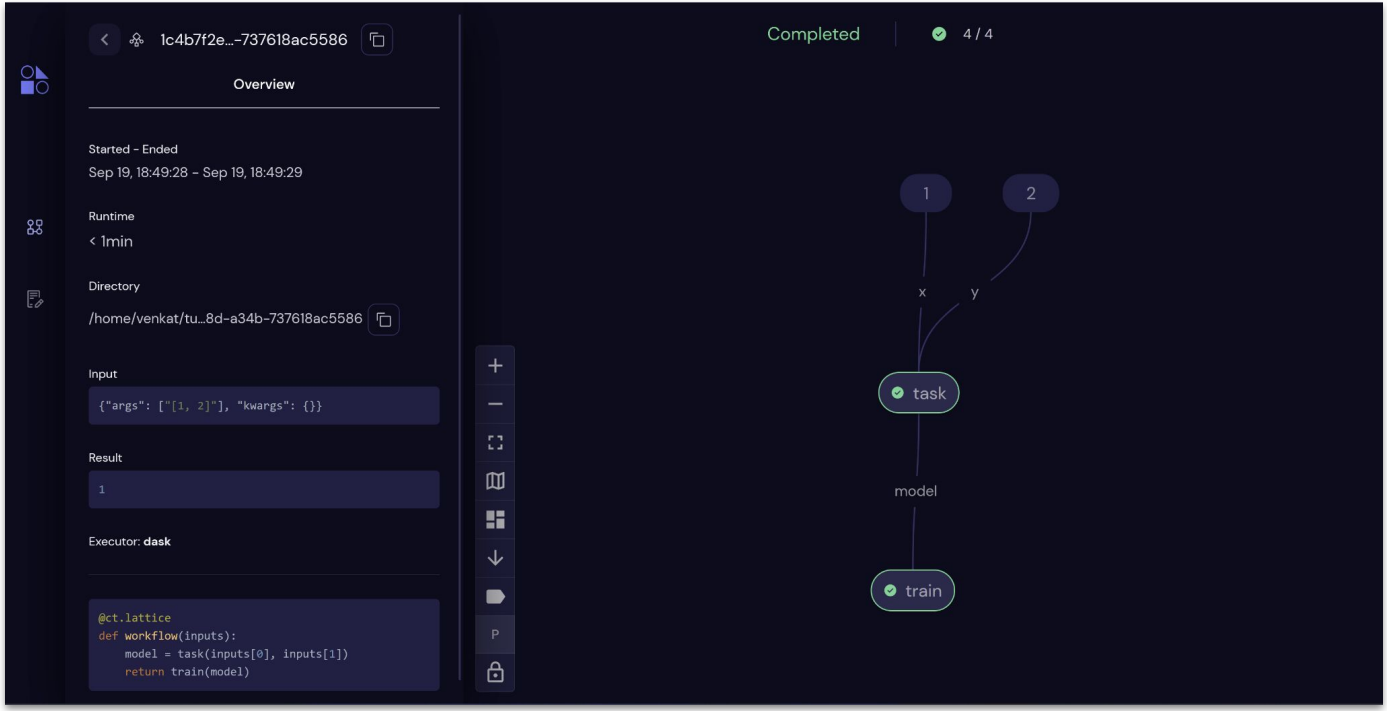


Minimal code changes required

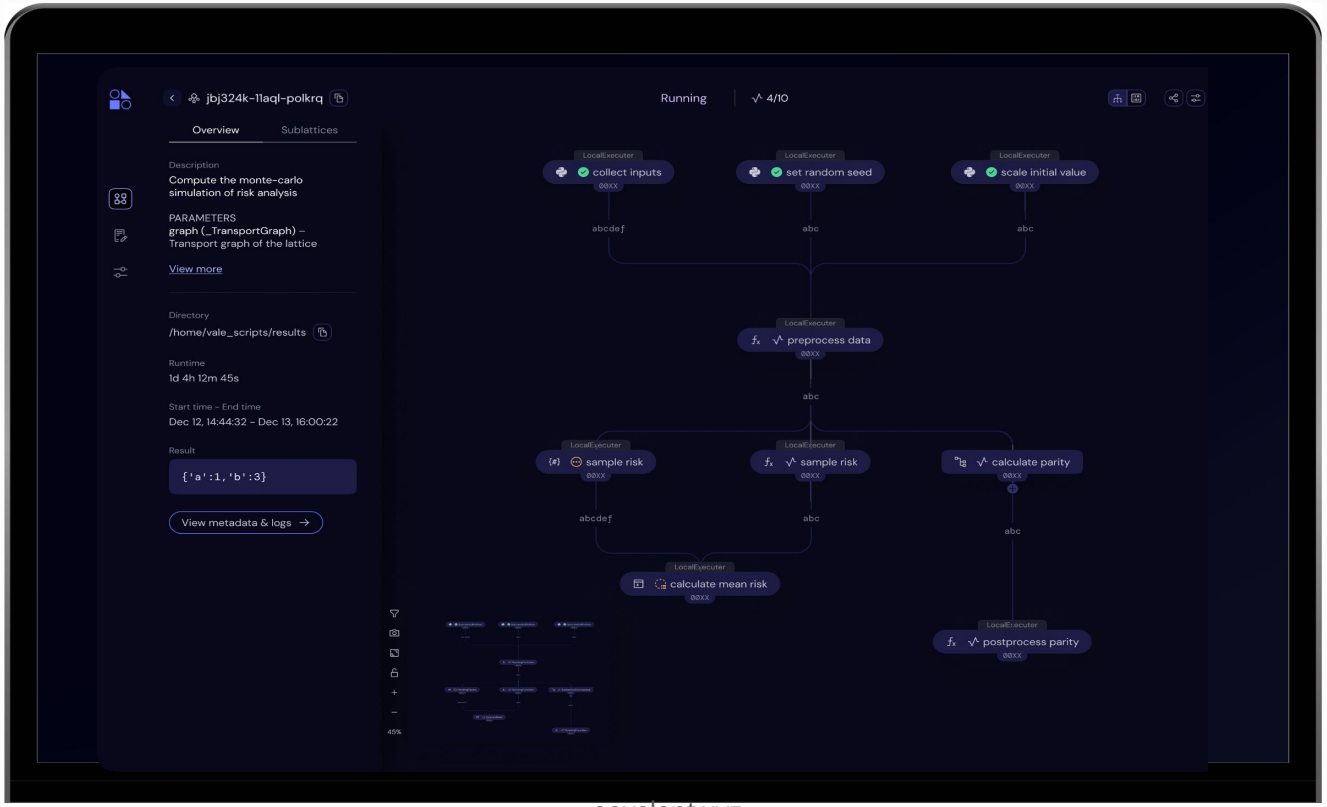
```python
def task(x, y):
    z = optimize(x, y)
    return z

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))
```

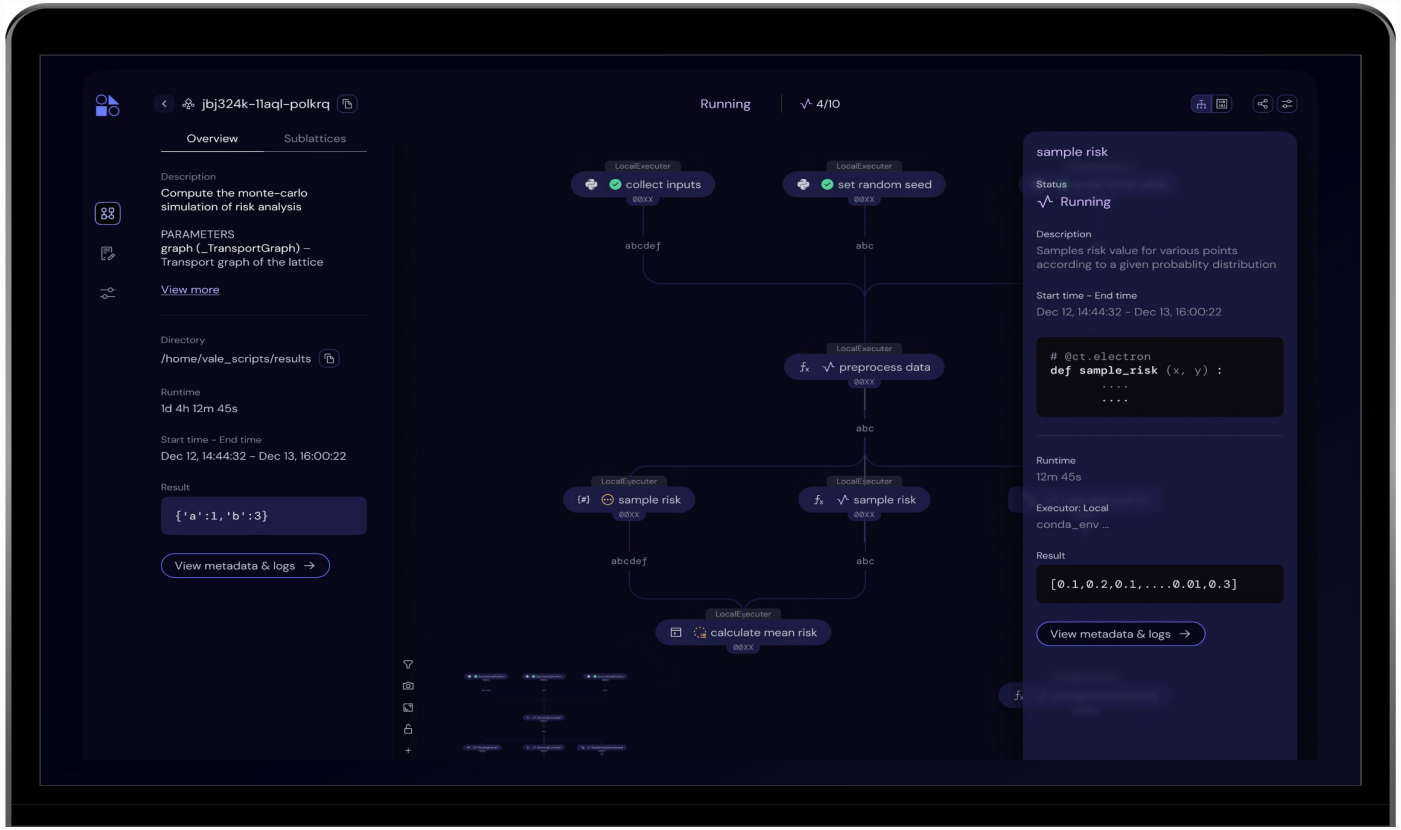
Covalent UI


1c4b7f2e...-737618ac5586
Overview
Started - Ended
Sep 19, 18:49:28 - Sep 19, 18:49:29
Runtime
< 1min
Directory
/home/venkat/tu...8d-a34b-737618ac5586
Input
{"args": ["[1, 2]"], "kwargs": {}}
Result
1
Executor: dask
@ct.lattice
def workflow(inputs):
model = task(inputs[0], inputs[1])
return train(model)
Completed
4 / 4
1
2
x
y
task
model
train

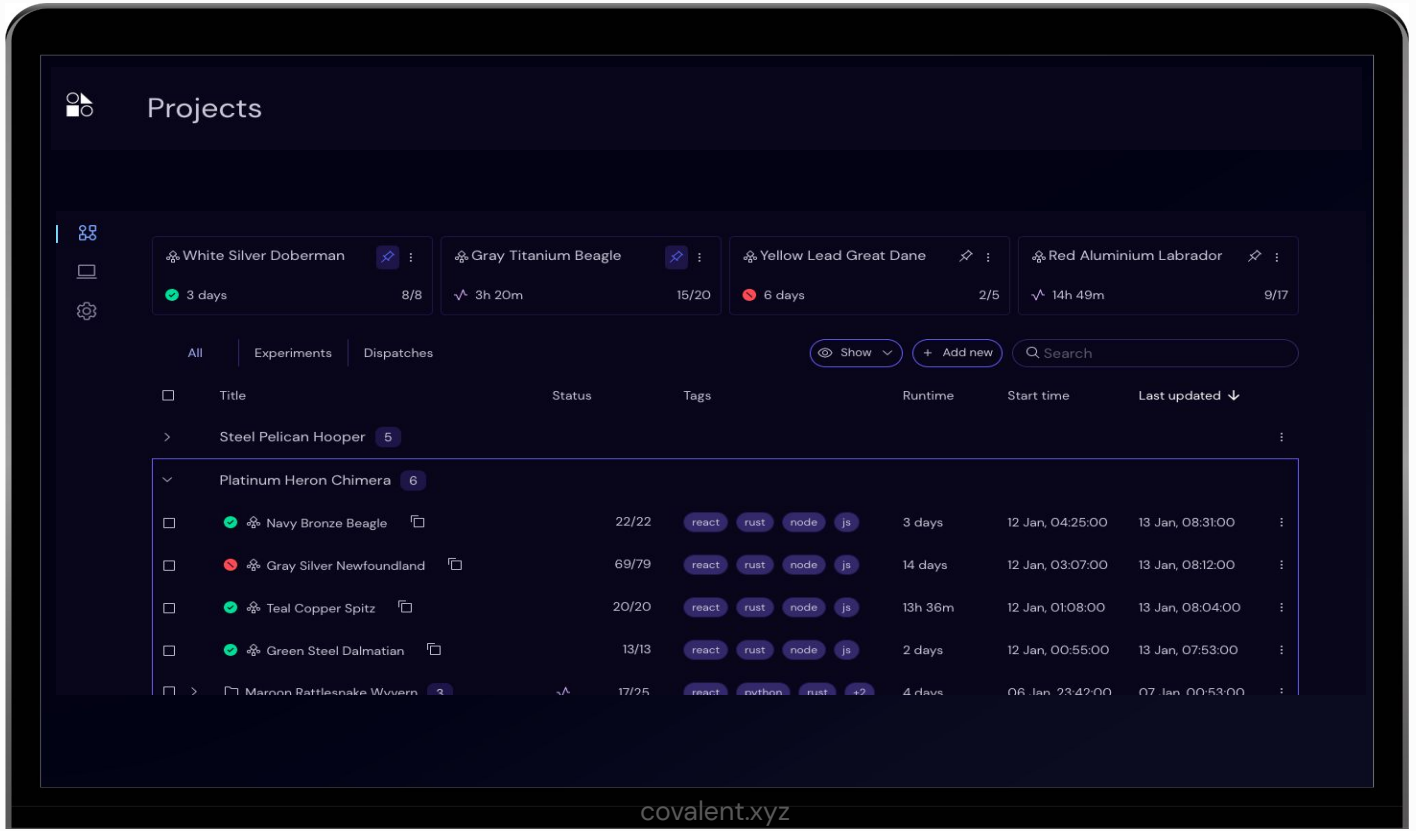
Visualize and monitor complex workflows



Inspect task-level results and metadata



Organize jobs across projects and experiments



You can contribute too!

Native parallelization

Pythonic workflows	Automatic checkpointing	Multiple language support	Little-to-no overhead	Customizable
Reproducibility	Code locally, run anywhere	Covalent. It's Open-Source!	Intuitive User-interface	Natively hybrid workflows
				Variety of executors
Code isolation	Parameter caching	Cloud agnostic	Interactive jobs	Start locally and scale

Code Samples

Example 1: Curve Fitting

SETUP

- Clone the Github repository

```
git clone https://github.com/AgnostiqHQ/tutorials_covalent_ieee_2022.git
```

- Download Miniconda
- Install the conda environment (`environment.yml` in root directory)

```
conda env create -f environment.yml
```

- Activate environment

```
conda activate ieee_covalent
```

- Add to kernel list

```
python -m ipykernel install --user --name=ieee_covalent
```

covalent start

covalent start --ignore-migrations

1. Functionalize your code

```
import numpy as np
import matplotlib.pyplot as plt
x = np.arange(10)
Y = np.random.random(10)

# 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])
```

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

def fit_xy(x,y,order):
    z = np.polyfit(x,y,order)
    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 cfit(x,y,order):
    x_new = np.linspace(x[0], x[-1], 50)
    fit=fit_xy(x,y,order)
    return plot_fit(x=x,y=y,xnew=x_new,fit=fit)
```

2. Decorate your functions

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

```python
import covalent as ct

@ct.electron
def fit_xy(x,y, order):
    z = np.polyfit(x, y, order)
    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 cfit(x, y, order):
  x_new = np.linspace(x[0], x[-1], 50)
  fit = fit_xy(x, y, order)
  return plot_fit(x, y, x_new, fit)
```

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

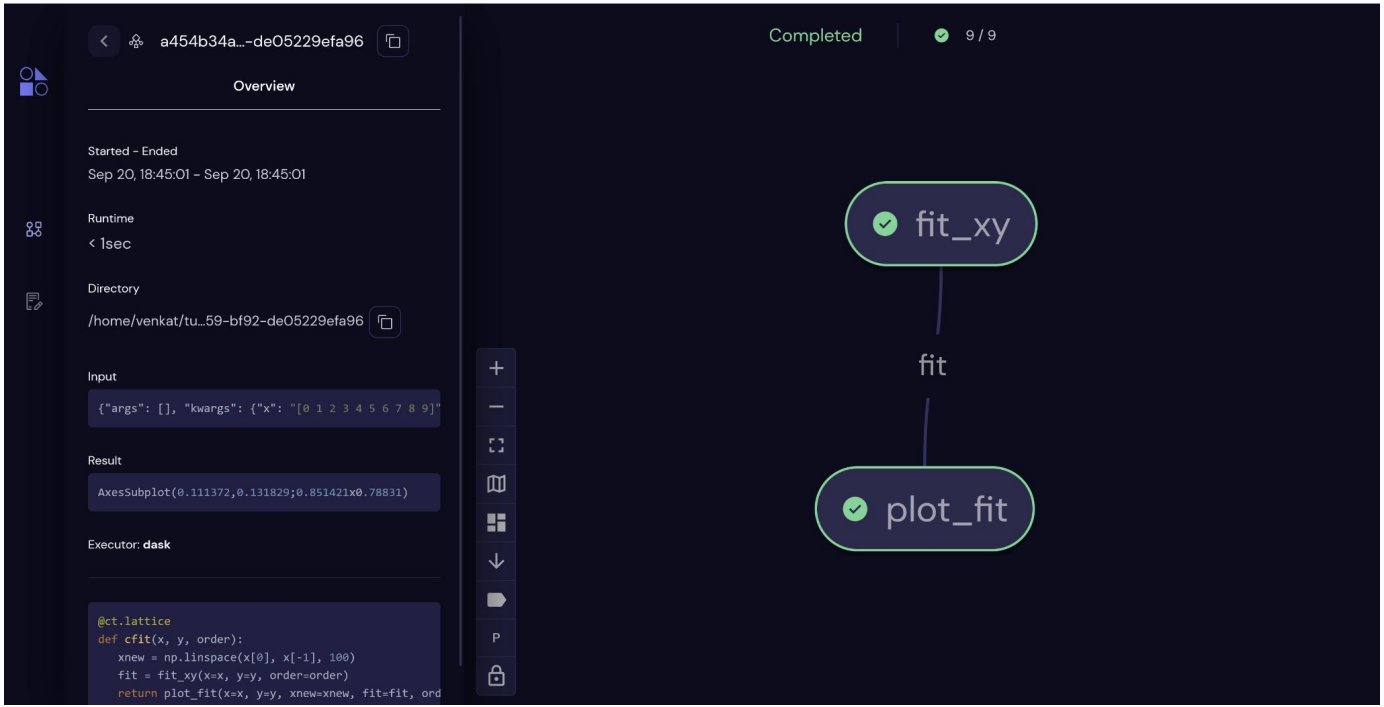
4. Dispatch your workflow

```
x = np.arange(10)
y = np.random.random(10)
dispatch_id = ct.dispatch(cfit)(x, y, 10)
```

5. Get the result

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

6. View results on UI



Internals

- Decorators declare intent
- **D**irected **A**cylic **G**raph (DAG) is built by sequentially inspecting the electrons (TransportGraph)
- Internally **Covalent** uses a local Dask cluster to execute the workflow
- Default cluster configuration can be set at covalent start time
  - Default workers = # of cores
  - Memory per workers = Total memory/Number of workers

Multiple Dispatches

Multiple dispatches

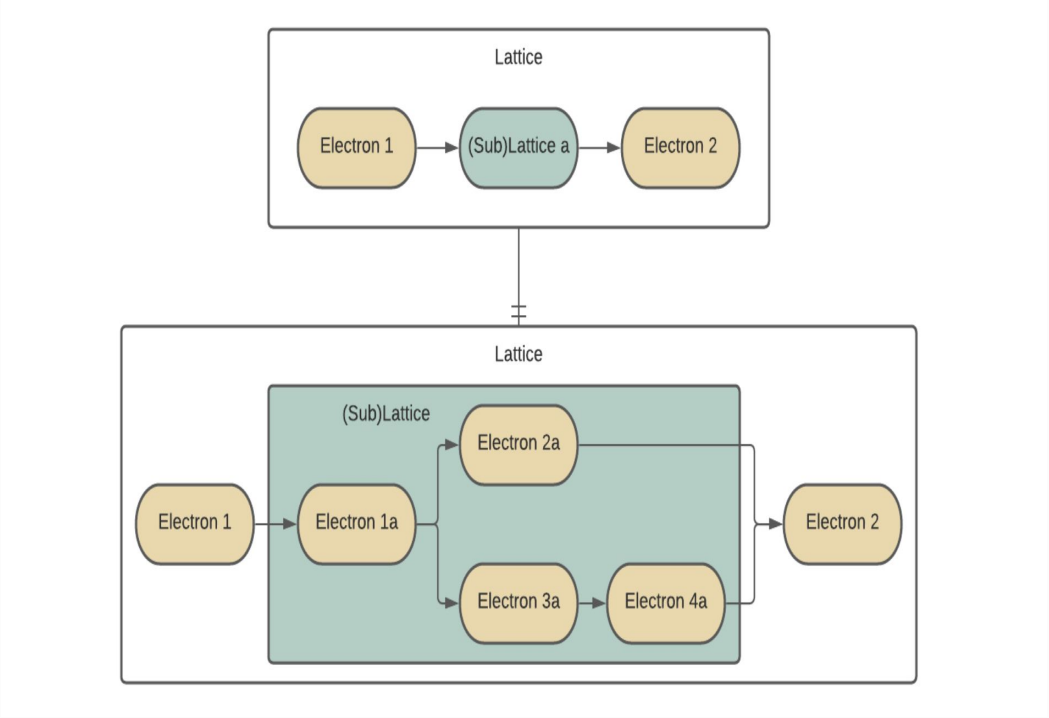
- Repeated experiments
- Large parameter sweeps, hyperparameter optimization in machine learning
- Dispatch a lattice with different parameters each time

```
inputs = [{"x": np.arange(N), "y": np.random.random(N), "order": np.random.randint(5, N)} for _ in range(5)]
dispatches = [ct.dispatch(exp)(input['x'], input['y'], input['order']) for input in inputs]
results = [ct.get_result(dispatch_id, wait=True) for dispatch_id in dispatches]
```

Sublattices

Sublattices

- A **lattice** converted into an **electron**
- Transforms a workflow into a node in a much bigger workflow
- Composition
- Makes creating larger workflows from existing ones much easier
- Add an electron **decorator** to an existing lattice

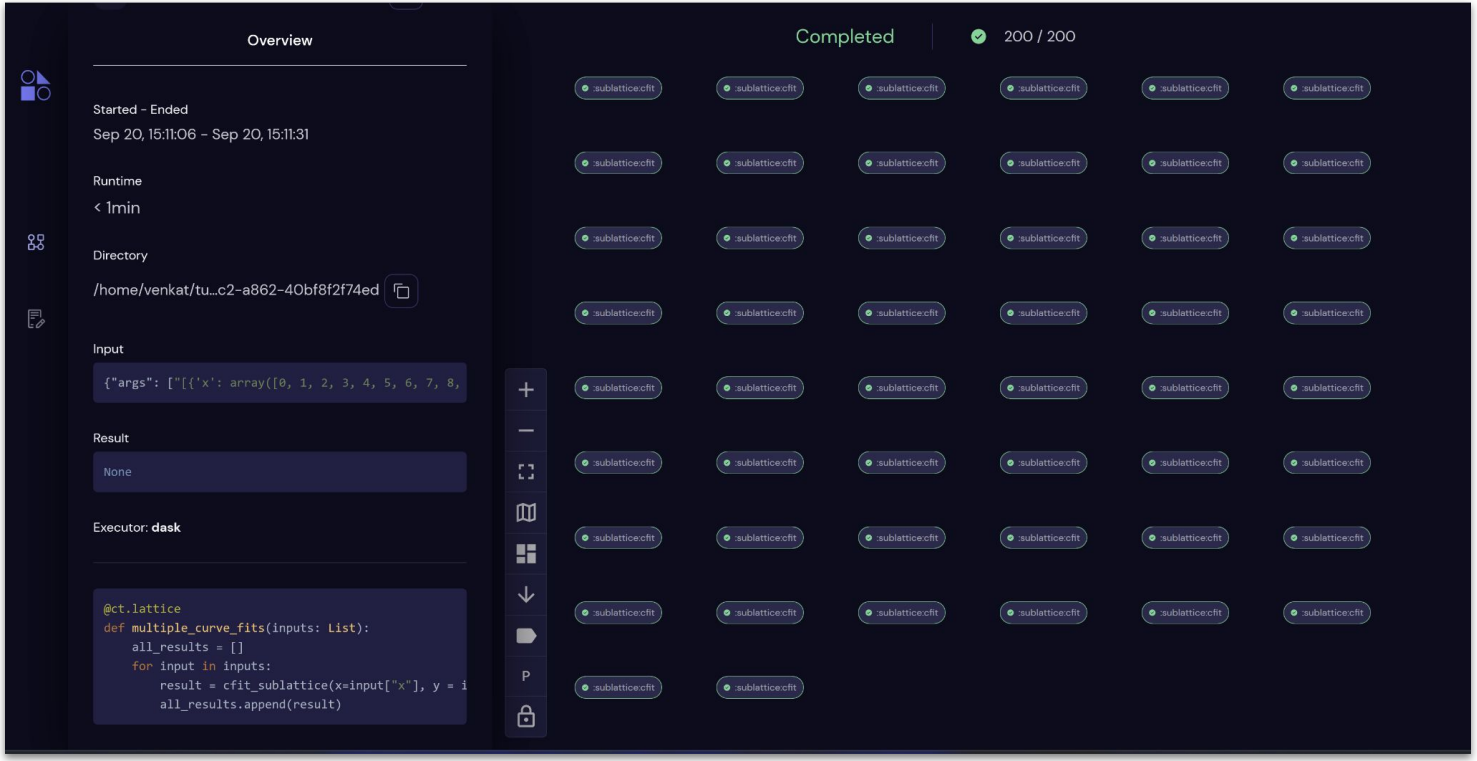


Motivation

- Nested experiments
- Created an experiment (lattice) consisting of several tasks (electrons)
- Dispatch the experiment to Covalent for execution with a specific set of parameters
- Create a new workflow that takes in a range of parameters as input
- Invoke the earlier workflow as a node
- Can recursively construct very large nested workflows

```
@ct.electron
def task1(*args, **kwargs):
    ...
    return result

@ct.electron
def task2(*args, **kwargs):
    ...
    return result

@ct.lattice
def experiment(**params):
    r1 = task1(*args, **kwargs)
    r2 = task2(*args, **kwargs)
    return r1 + r2

# Create sub-lattice
run_experiment = ct.electron(experiment)

@ct.lattice
def experiment_suite(**parameters):
    for param in parameters
        run_experiment(**param)
    return
```

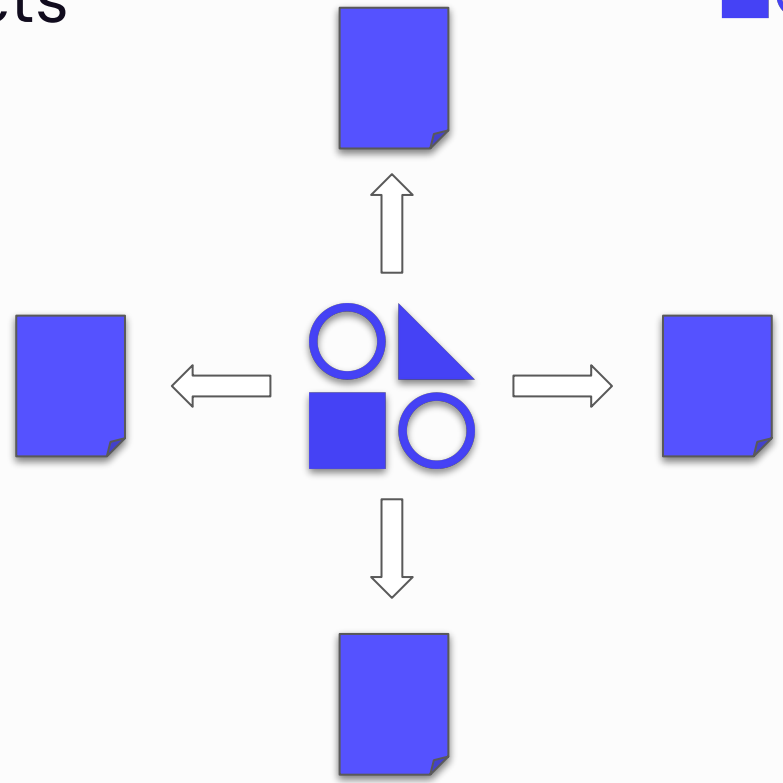
Sublattice


Overview
Started - Ended
Sep 20, 15:11:06 - Sep 20, 15:11:31
Runtime
< 1min
Directory
/home/venkat/tu...c2-a862-40bf8f2f74ed
Input
{"args": ["[{'x': array([0, 1, 2, 3, 4, 5, 6, 7, 8,
Result
None
Executor: dask
@ct.lattice
def multiple_curve_fits(inputs: List):
all_results = []
for input in inputs:
result = cfit_sublattice(x=input["x"], y = i
all_results.append(result)
Completed
200 / 200
:sublattice:cfit

Metadata

Results Manager/Result Objects

- Storing, updating and retrieving workflow results
- Decoupled from dispatch server
- Users can query and track workflow progress without needing source code
- Experiment outcomes can be safely stored
- Build in checkpointing
- Easy API to retrieve result



Code Samples

Example 2: Eigenvalues

Matrix Eigenvalues

- Important in classical/quantum mechanics
- Can be computationally heavy
- Demonstrate concept of **remote executors** in Covalent
- Can display beautiful symmetries!

Matrix Eigenvalues

```
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]])
```

Compute the eigenvalues of a lot of such matrices and view the results

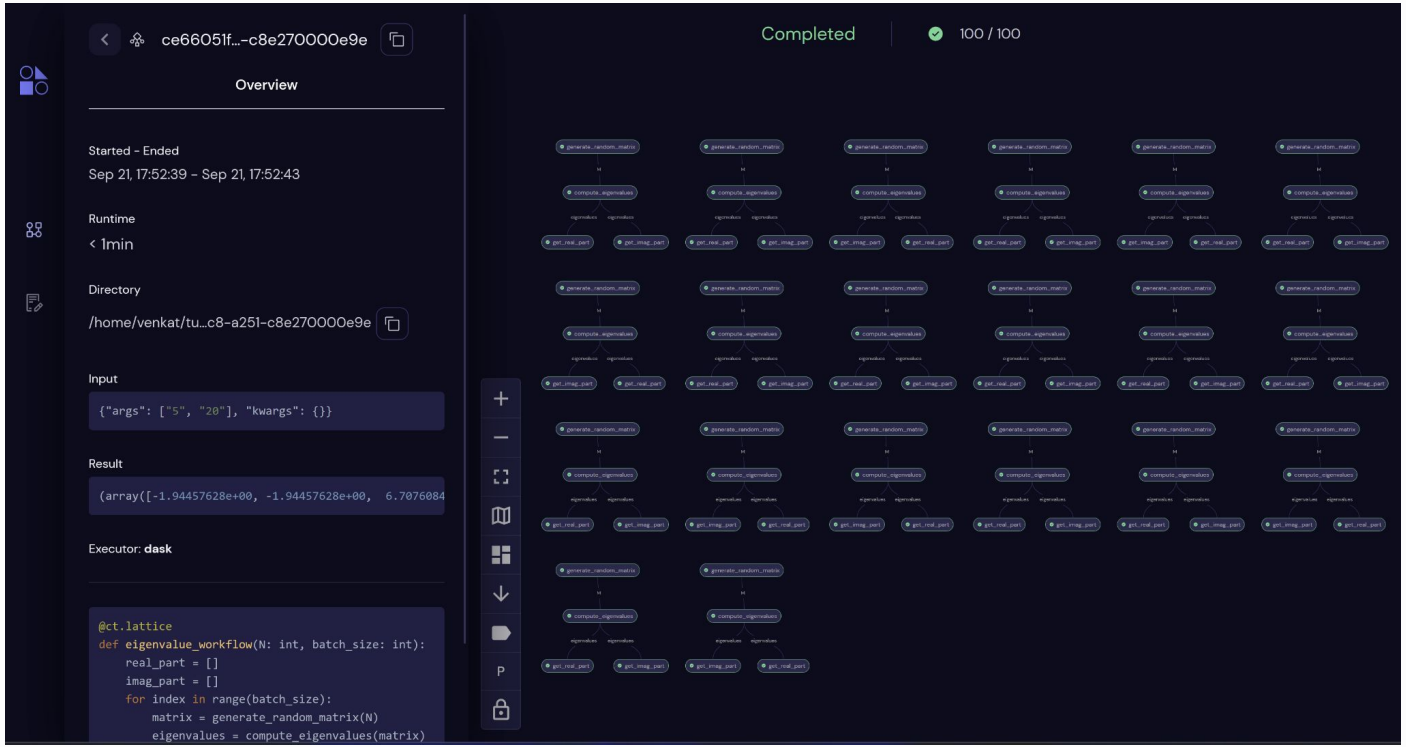
Workflow Electrons

```
@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]
```

Lattice

```python
@ct.lattice
def eigenvalue_workflow(N: int, batch_size: int):
    real_part = []
    imag_part = []
    for index in range(batch_size):
        matrix = generate_random_matrix(N)
        eigenvalues = compute_eigenvalues(matrix)
        real_part.append(get_real_part(eigenvalues))
        imag_part.append(get_imag_part(eigenvalues))
    return np.asarray(real_part).flatten(), np.asarray(imag_part).flatten()
```

Workflow Graph



Executors

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 dictate the execution environment/platform of a particular task in the workflow
- Executors are plugins to Covalent and can be installed separately via **pip**
- Supported executors
    - Local, Dask, SSH, SLURM, AWS (EKS, ECS, Batch, Lambda, Braket), Qiskit Runtime (in development)

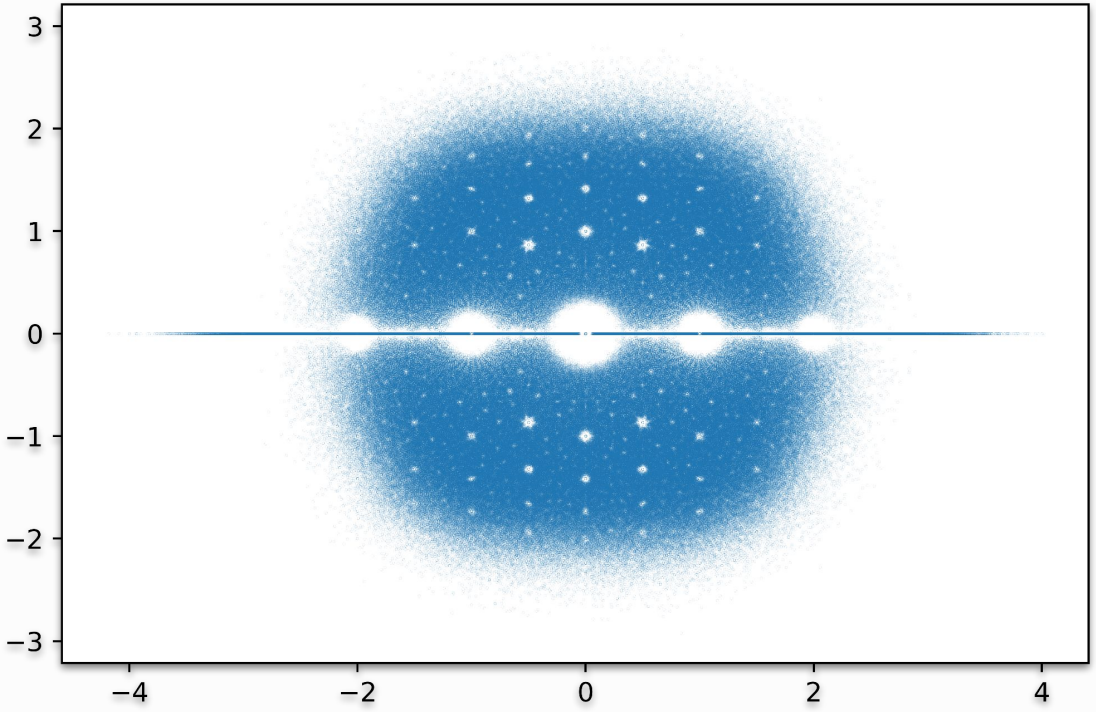
SSHExecutor

```
pip install covalent-ssh-plugin
```

SSHExecutor

```
from covalent.executor import SSHExecutor

ssh = SSHExecutor(username="<username>",hostname="<hostname>",
                  ssh_key_file="<ssh private key>")

@ct.electron(executor=ssh)
def compute_heavy_task(*args, **kwargs):
    ...
    return result

@ct.lattice
def workflow(*args, **kwargs):
    ...
    res = compute_heavy_task(*args, **kwargs)
    ...
    return res
```

Result



SLURMExecutor

```
pip install covalent-slurm-plugin
```

SLURM Executor

```python
from covalent.executor import SlurmExecutor

slurm = SlurmExecutor(username="xyz", address="cluster.example.edu",
            ssh_key_file="~/.ssh/ssh_private_key", options={"ntasks": 1,"cpus-per-task": 5,
            "partition": "compute", "nodelist": "node001"})

@ct.electron(executor=slurm)
def compute_heavy_task(*args, **kwargs):
    ...
    return result

@ct.lattice
def workflow(*args, **kwargs):
    res = compute_heavy_task(*args, **kwargs)
    return res
```

AWS Executor Plugins

AWS Executor Plugins

- Covalent supports a wide range of AWS specific cloud executors
- These executors allow users to dispatch electrons to different AWS compute services
- Supported AWS services
  - AWS Batch
  - AWS Lambda
  - AWS EC2
  - AWS Braket
  - AWS ECS/Fargate
  - AWS EKS

AWS Executors

- AWS Braket
  - Suitable for hybrid workflows (mix of classical and quantum resources)
- AWS Lambda
  - Embarrassingly parallel workflows, several short lived electrons (light weight)
- AWS ECS
  - Useful for moderate to heavy workloads (low memory requirements)
- AWS Batch
  - Useful for heavy compute workloads (high CPU/memory)
- AWS EC2
  - General purpose compute workloads, can be tuned to workflow's compute requirements

AWS Lambda Executor

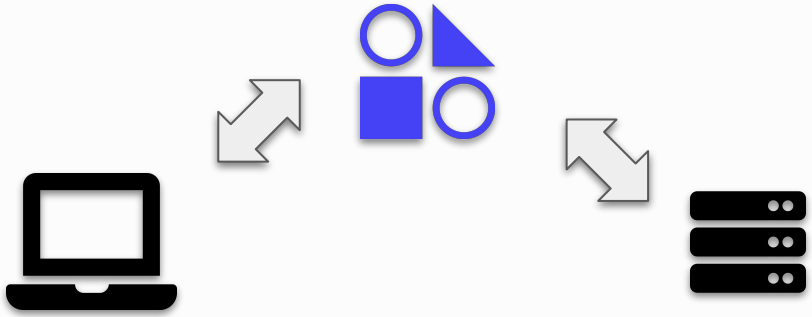
```
import covalent as ct
from covalent.executor import AWSLambdaExecutor

awslambda = AWSLambdaExecutor(credentials="~/.aws/credentials",
                                    profile="default",
                                    region="us-east-1",
                                    lambda_role_name="CovalentLambdaExecutionRole",
                                    s3_bucket_name="covalent-lambda-job-resources")

@ct.electron(executor=awslambda)
def task(*args, **kwargs):
    ...
    return result


@ct.lattice
def workflow(*args, **kwargs):
    res = task(*args, **kwargs)
    return res
```

Fully configurable!!

Extra features

File transfers

- Transfer file from local to remote machine (vice-versa)
- AWS S3 buckets supported
- pre/post execution to upload/download files
- Specified as electron decorator arguments



File Transfer API

```python
import covalent as ct

@ct.electron(
    files=[ct.fs.FileTransfer('/home/ubuntu/src_file', '/home/ubuntu/dest_file')]
)
def my_task(files=[]):
    from_file, to_file = files[0]
    with open('/home/ubuntu/dest_file', 'r') as f:
        return f.read()

@ct.lattice
def file_transfer_workflow():
    return my_task()

# Dispatch the workflow
dispatch_id = ct.dispatch(file_transfer_workflow)()
```

Electron dependencies (Deps)

- `Deps` provide a way to specify the electron level dependencies in Covalent
- `DepsPip`
  - Install any Python libraries required by an **electron**
- `DepsBash`
  - Execute any **Bash** command/script before/after electron execution
- `DepsCall`
  - Invoke any Python function before/after execution of an electron

Electron dependencies

```
import covalent as ct
from covalent import DepsPip, DepsBash, DepsCall

def execute_before_electron(a, b):
...

def shutdown_after_electron():
...

@ct.electron(
    deps_pip=DepsPip(packages=["numpy==0.23", "qiskit"]),
    deps_bash=DepsBash(commands=["echo $PATH", "ssh foo@bar.com"]),
    call_before=DepsCall(execute_before_electron, args=(1, 2)),
    call_after=DepsCall(shutdown_after_electron),
    executor=slurm
)
def task(*args, **kwargs):
    ...
```

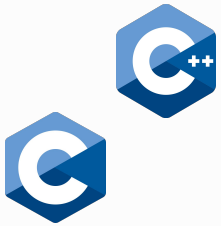
C/C++ support

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/C++ through Covalent
- More language support coming
  - Julia
  - Fortran
  - (Request your favorite language on GitHub)

Why is it important?

- Interfacing with legacy code
- Quantum simulations on classical hardware expensive (# of qubits)
- A lot of quantum simulators are available in C/C++
  - NVIDIA cuQuantum, QODA
    - GPU acceleration for quantum simulations
  - Intel Quantum Simulator (IQS) (C/C++)
    - Accelerate quantum simulations on Intel CPUs
  - Quantum++ (C++ 11)
  - Qrack (C++/OpenCL)



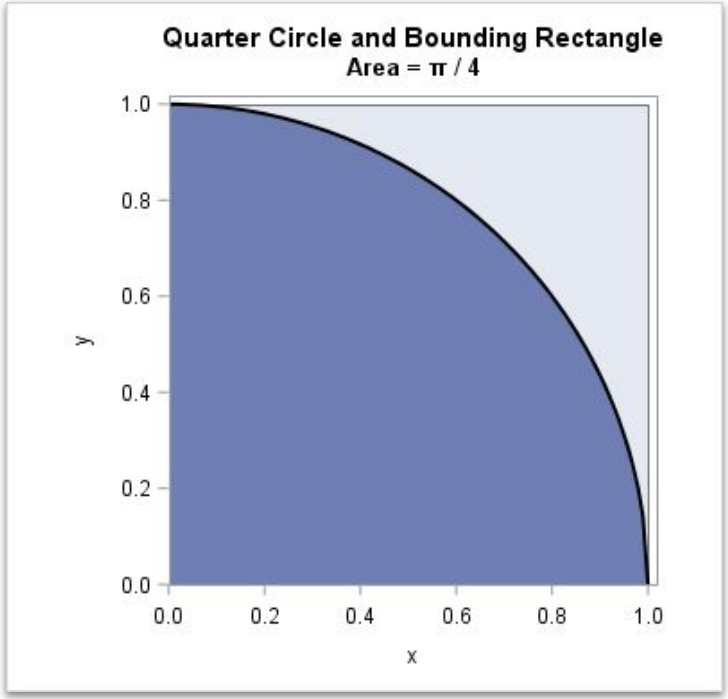
Code Samples

Example 3: Estimate PI in C

Interfacing with C

- CPython (most widely used) is written purely in C
- Python module written in C
- Compile the module to a shared library
- Use the functions defined in the module in **Covalent** workflows
- Python/C API
- Wrappers like SWIG can also be leveraged

Estimate PI: Area under the curve



Python/C API

```c
#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

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

static struct PyModuleDef cpiapprox = {
    PyModuleDef_HEAD_INIT,
    "cpiapprox",    /* 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. */
    PiapproxMethods
};

PyMODINIT_FUNC PyInit_cpiapprox(void) {
    return PyModule_Create(&cpiapprox);
}
```

Build and Install

- setup.py

```
from distutils.core import setup, Extension

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

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

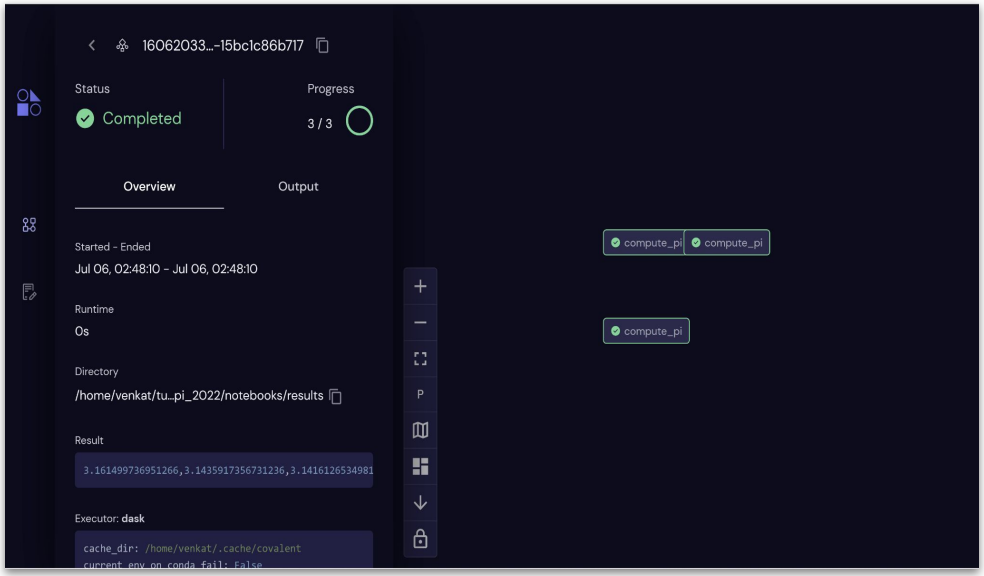
- Build/install the module

```
python setup.py install
```

Covalent workflow

```python
import covalent as ct
import cpiapprox

@ct.electron
def compute_pi(n):
    return cpiapprox.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]
```



Code Samples

Example 4: Vector Arthimetic in C++

Interfacing with C++

- Pybind11 is a lightweight header-only library that exposes C++ types in Python and vice-versa
- It support a lot of C++ core features
  - Functions (passing by value and reference)
  - Instance/static method
  - Function overloading
  - Templates
  - STL types
  - Iterators, ranges
  - single/multiple inheritance

C++ with Covalent

- As a very minimal example of, we write a simple vector arithmetic library in C++ and expose it as a Python package to Covalent via **pybind11**
- We use core C++ STL types (std::vector) and STL algorithms to implement three functions
  - `Vecadd`
    - Add two **std::vector** of type double (elementwise)
  - `Vecmul`
    - Multiply two **std::vector** of type double (elementwise)
  - `Vecdiv`
    - Divide two **std::vector** of type double (elementwise)

Vector arithmetic

```cpp
#include <vector>
#include <pybind11/pybind11>
#include <numeric>
#include <algorithm>
#include <functional>

using dvec = std::vector<double>;

dvec vecadd(const dvec& a, const dvec& b) {
    dvec c;
    std::transform(a.begin(), a.end(), b.begin(),
       std::back_inserter(c), std::plus<double>());
    return c;
}
dvec vecmul(const dvec& a, const dvec& b) {
    dvec c;
    std::transform(a.begin(), a.end(), b.begin(),
        std::back_inserter(c), std::multiplies<double>());
    return c;
}
dvec vecdiv(const dvec& a, const dvec& b) {
    dvec c;
    std::transform(a.begin(), a.end(), b.begin(),
        std::back_inserter(c), std::divides<double>());
    return c;
}
```

Build Python interface

```
// main.cc
PYBIND11_MODULE(cpparthimetic, m) {
    m.def("vecadd", &vecadd, "Add two python lists");
    m.def("vecmul", &vecmul, "Multiply two python lists");
    m.def("vecdiv", &vecdiv, "Divide two python lists elementwise");
}
```

Compiling C++ to generate module

- setup.py

```
# setup.py
from setuptools import setup
from pybind11.setup_helpers import Pybind11Extension, build_ext

cpparthimetic_module = Pybind11Extension('cpparthimetic',
sources=['main.cc'])

setup(name = 'cpparthimetic',
    version='0.1.0',
    author = 'venkat Bala',
    author_email="venkat@agnostiq.ai",
    cmdclass={"build_ext": build_ext},
    ext_modules=[cpparthimetic_module])
```

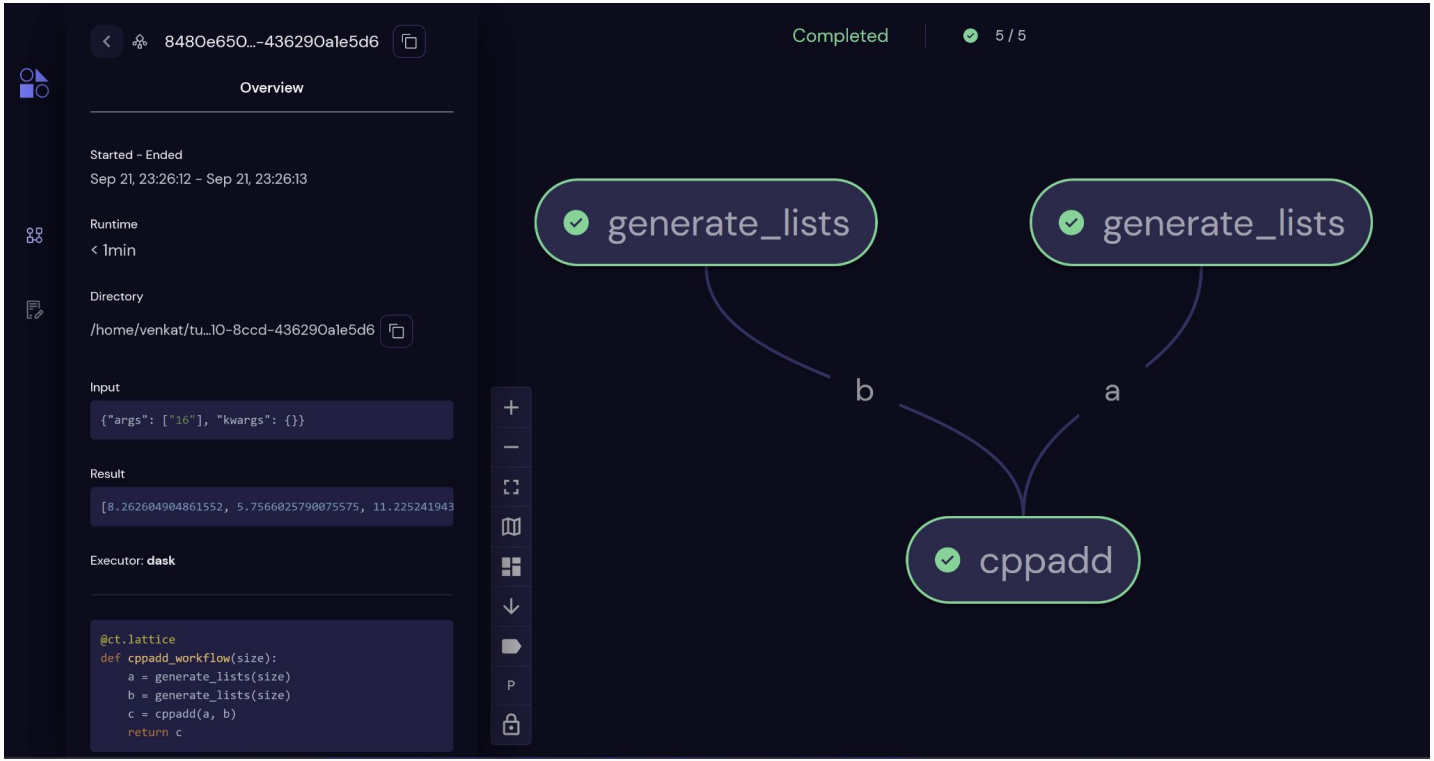
- Build/install module

```
python setup.py install
```

Covalent workflow

```python
import covalent as ct
import random
import cpparthimetic

@ct.electron
def generate_list(N: int):
    return [random.uniform(0.1, 1) for i in range(N)]

@ct.electron
def cppadd(a: List[float], b: List[float]):
    return cpparthimetic.vecadd(a, b)

@ct.lattice
def cppadd_workflow(size):
    a = generate_lists(size)
    b = generate_lists(size)
    return cppadd(a, b)

dispatch_id = ct.dispatch(cppadd_workflow)(2**20)
add_result = ct.get_result(dispatch_id, wait=True)
```

C++ workflow


8480e650...-436290a1e5d6
Overview
Started - Ended
Sep 21, 23:26:12 - Sep 21, 23:26:13
Runtime
< 1min
Directory
/home/venkat/tu...10-8ccd-436290a1e5d6
Input
{"args": ["16"], "kwargs": {}}
Result
[8.262604904861552, 5.7566025790075575, 11.225241943
Executor: dask
@ct.lattice
def cppadd_workflow(size):
a = generate_lists(size)
b = generate_lists(size)
c = cppadd(a, b)
return c
Completed
5 / 5
generate_lists
generate_lists
b
a
cppadd

Summary

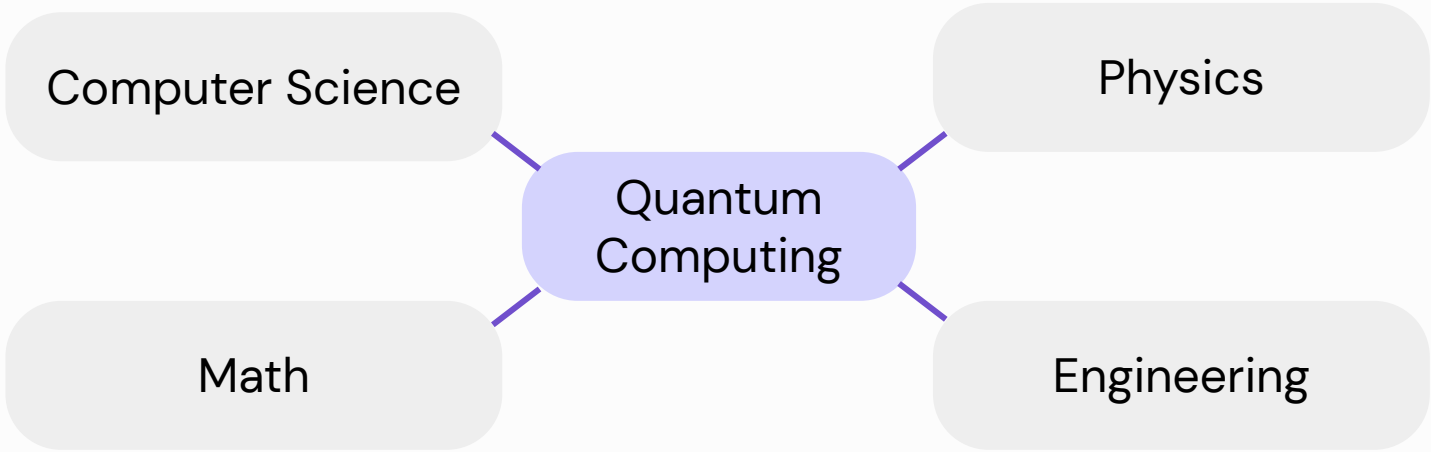
Covalent

- Covalent is fully open source and under going active development
- Contributions, feature requests are welcomed
- Source code
  - https://github.com/AgnostiqHQ/covalent
- Documentation
  - https://covalent.readthedocs.io/en/stable/
- Upcoming features/issues/bug reports
  - https://github.com/AgnostiqHQ/covalent/issues
- Latest/pre-releases available on PyPI
  - https://pypi.org/project/covalent/

AgnostiqHQ/covalent

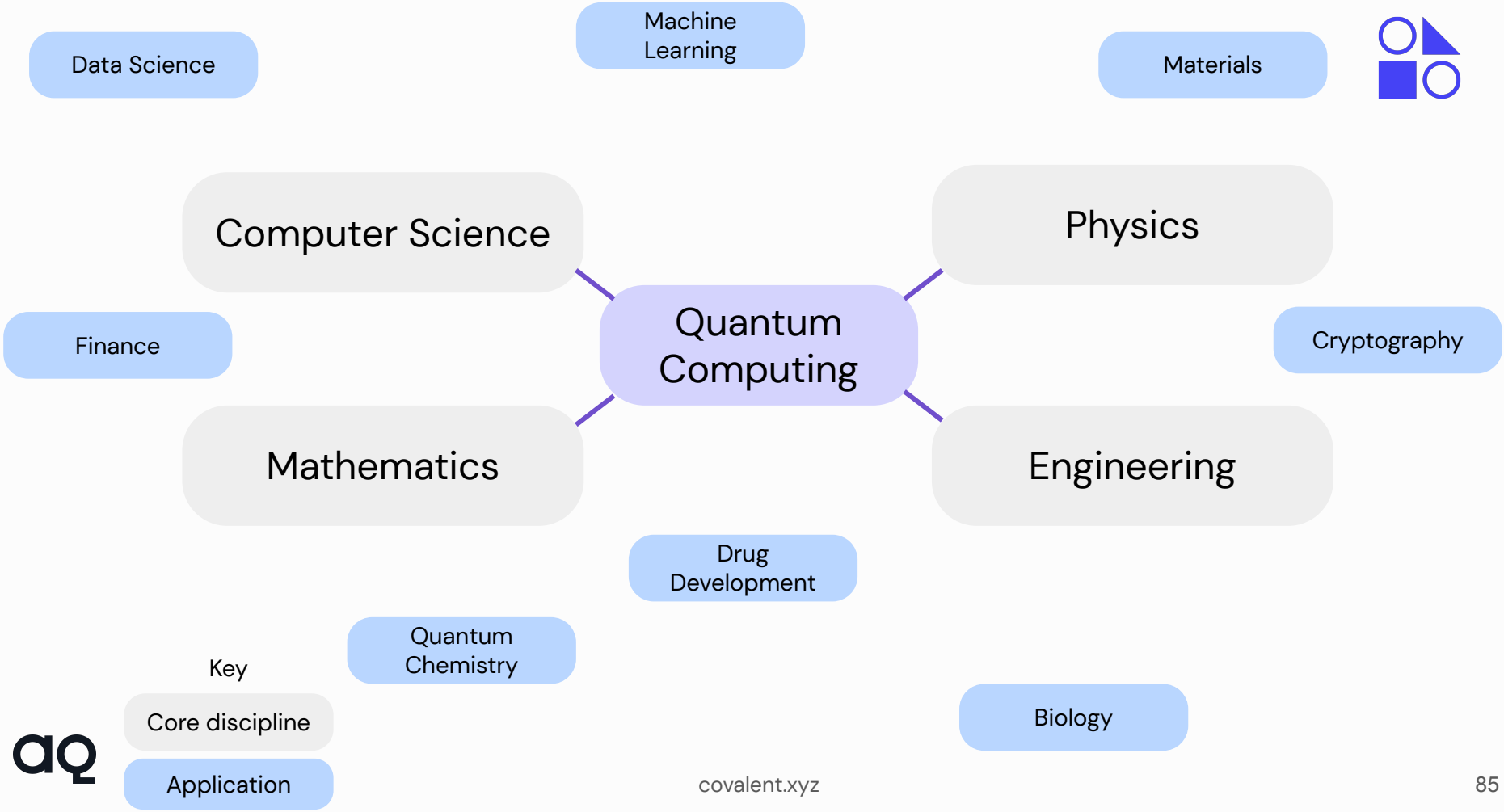
```
pip install covalent
```

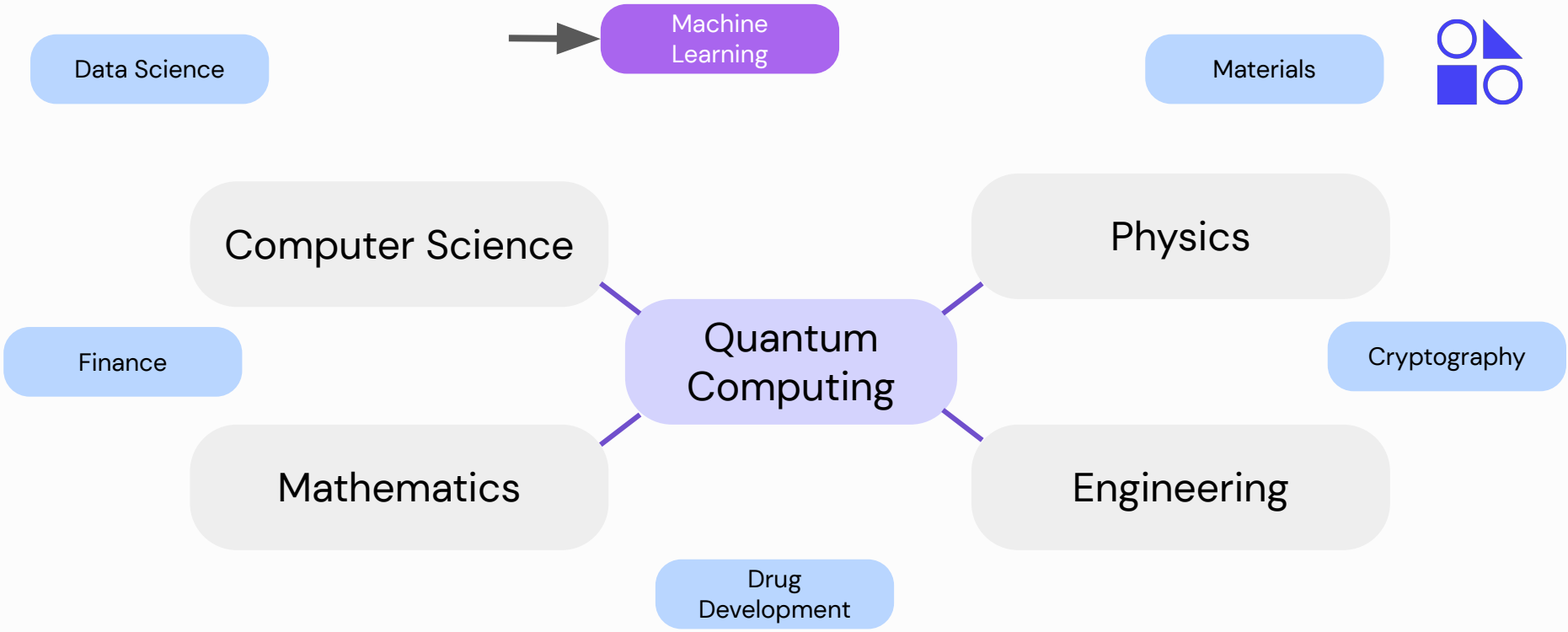
A (very brief) Introduction to Quantum Machine Learning



Key

Core discipline


Data Science
Machine Learning
Materials
Computer Science
Physics
Finance
Quantum Computing
Cryptography
Mathematics
Engineering
Drug Development
Quantum Chemistry
Key
Core discipline
Application
Biology


Data Science
Machine Learning
Materials
Computer Science
Physics
Finance
Quantum Computing
Cryptography
Mathematics
Engineering
Drug Development
Quantum Chemistry
Key
Core discipline
Application
Biology

What is ~~Quantum~~ Machine Learning?

- Before addressing Quantum Machine Learning (QML), we must first address ML
- Model parameters are tuned while being fed a training dataset
- The predictive capability of a model is assessed using a testing dataset

What is ~~Quantum~~ Machine Learning?

- Before addressing Quantum Machine Learning (QML), we must first address ML
- Model parameters are tuned while being fed a training dataset
- The predictive capability of a model is assessed using a testing dataset

ML: The construction of predictive models without *explicit instruction*.

What is ~~Quantum~~ Machine Learning?

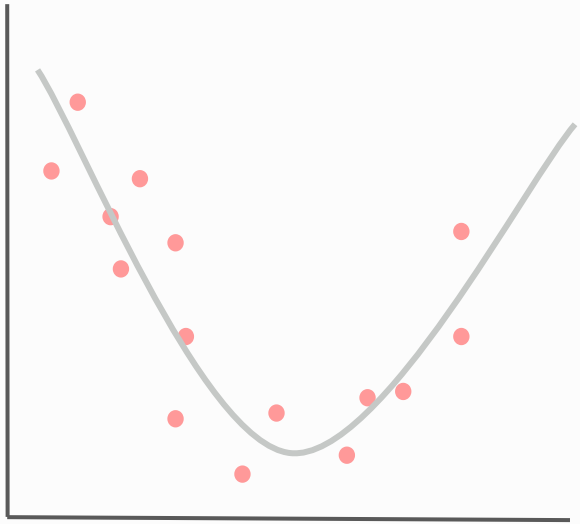
- Before addressing Quantum Machine Learning (QML), we must first address ML
- Model parameters are tuned while being fed a training dataset
- The predictive capability of a model is assessed using a testing dataset

ML: The construction of predictive models without *explicit instruction*.

Often, validation is performed between training and testing

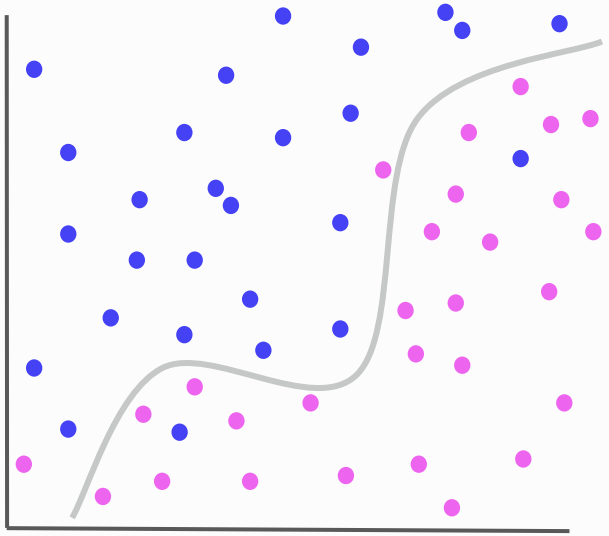
What is ~~Quantum~~ Machine Learning?

Regression



Logistic regression, random forest...

Classification



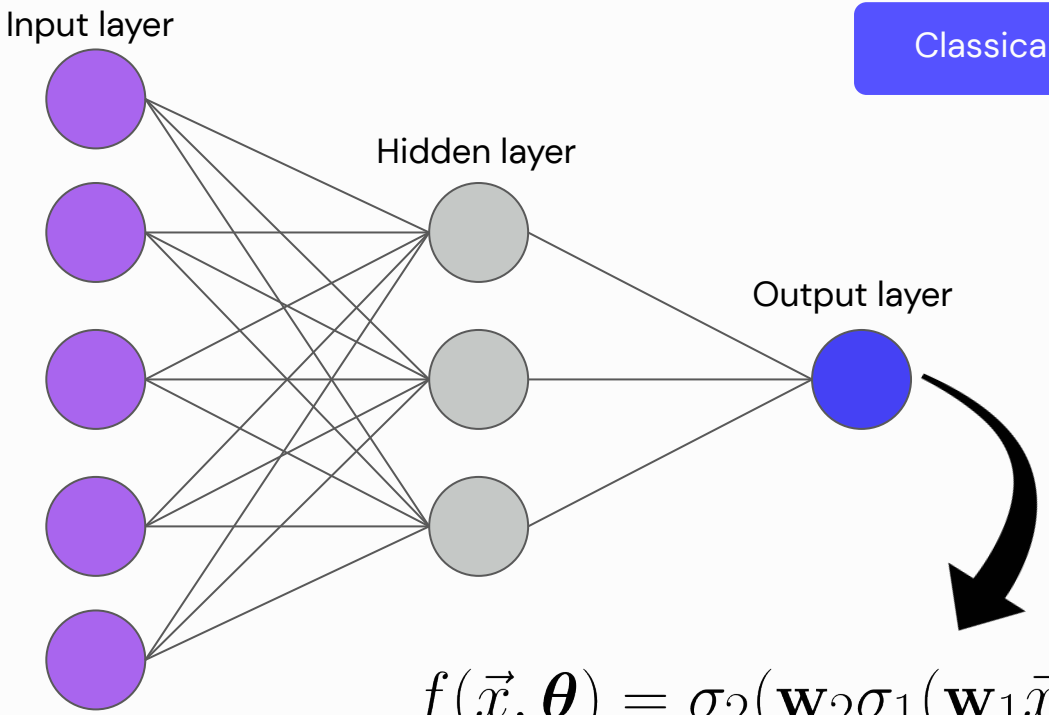
Support vector machines, autoencoders....

*Just two of the most common applications

What is ~~Quantum~~ Machine Learning?

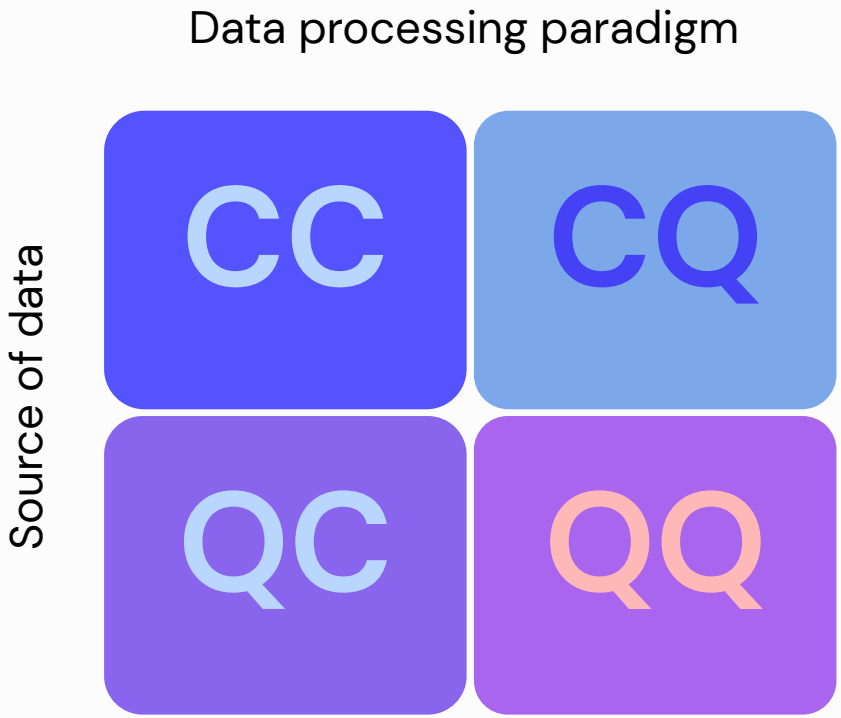
Classical neural networks

$$\vec{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_m \end{bmatrix}$$

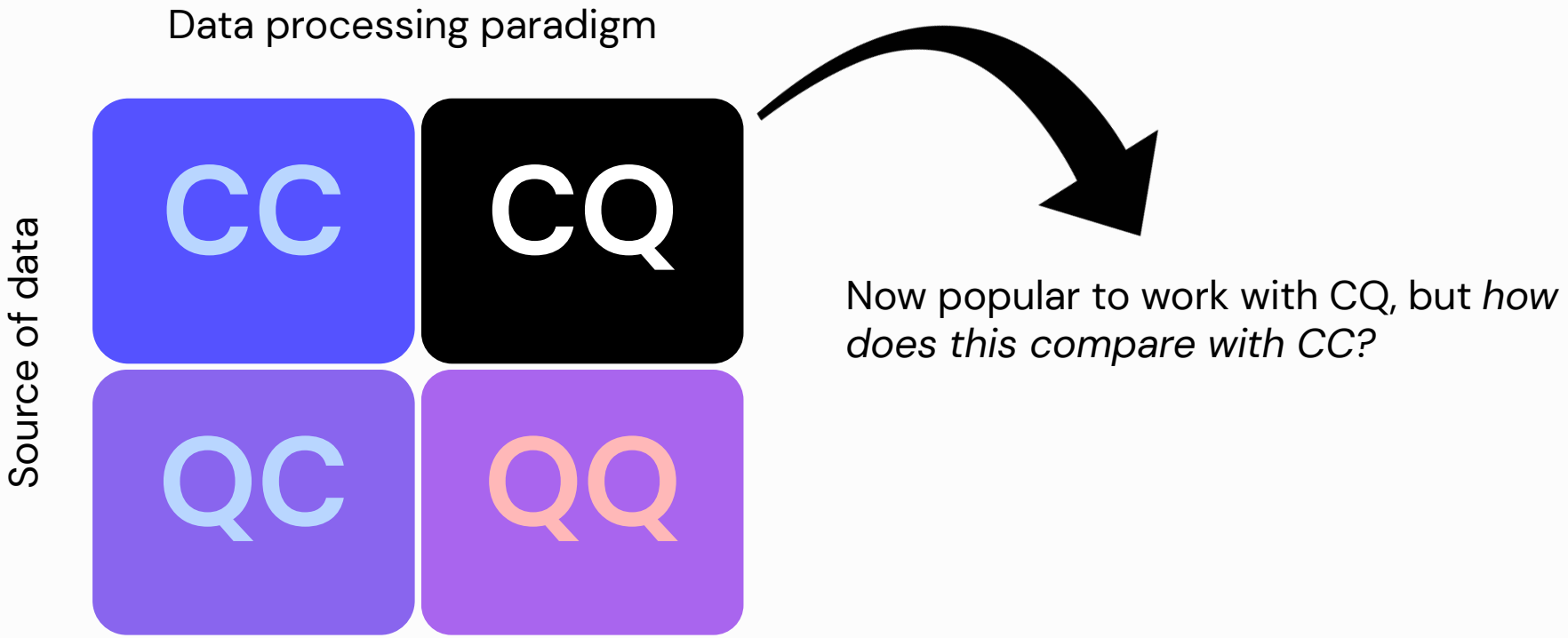


$$f(\vec{x}, \boldsymbol{\theta}) = \sigma_2(\mathbf{w}_2\sigma_1(\mathbf{w}_1\vec{x} + b_1) + b_2)$$

What is Quantum Machine Learning?



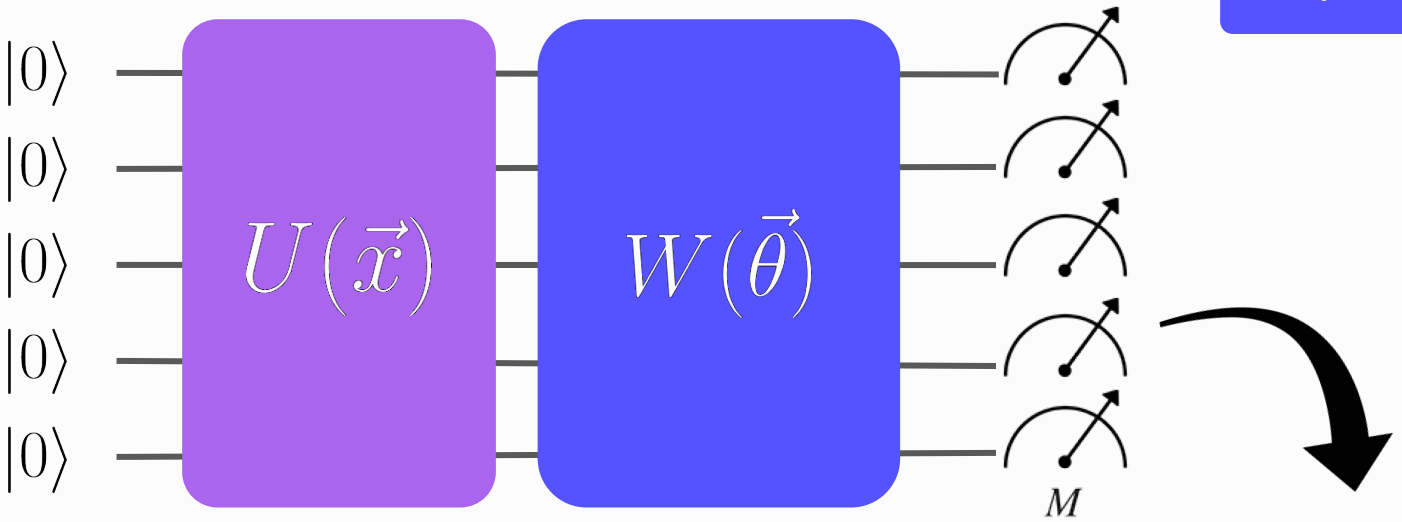
What is Quantum Machine Learning?



Now popular to work with CQ, but *how does this compare with CC?*

What is Quantum Machine Learning?

Quantum neural networks



$$f(\vec{x},\vec{\theta}) = \langle 0\ldots|U^{\dagger}(\vec{x})W^{\dagger}(\vec{\theta})HW(\vec{\theta})U(\vec{x})|0\ldots\rangle$$

Classical and Quantum Similarity Learning

What is similarity learning?

Similarity learning is the act of learning the notion of what makes two (or more) object similar

One way of handling this classically is to train a Siamese network. Basically two neural networks.

To handle this quantumly, we can use quantum similarity networks, or, GQSim

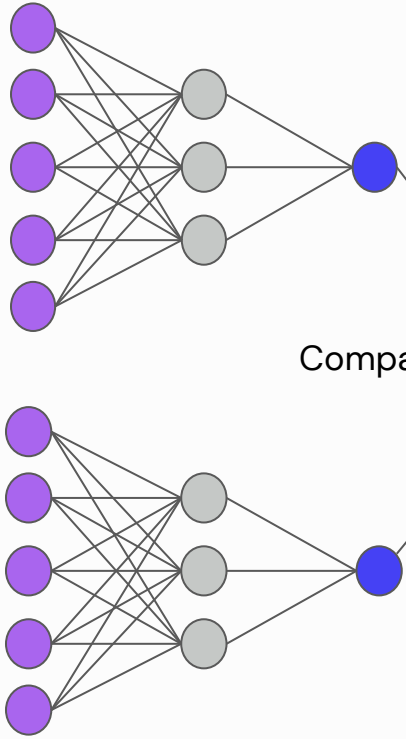
$$f(\vec{x}_i, \vec{x}_j, \vec{\theta}) \approx y_{ij}$$

$\vec{x}_i$	$\vec{x}_j$	y_{ij}
		0 (similar)
		1 (dissimilar)

Classical Siamese Network

$\vec{x}_i$

$\vec{x}_j$

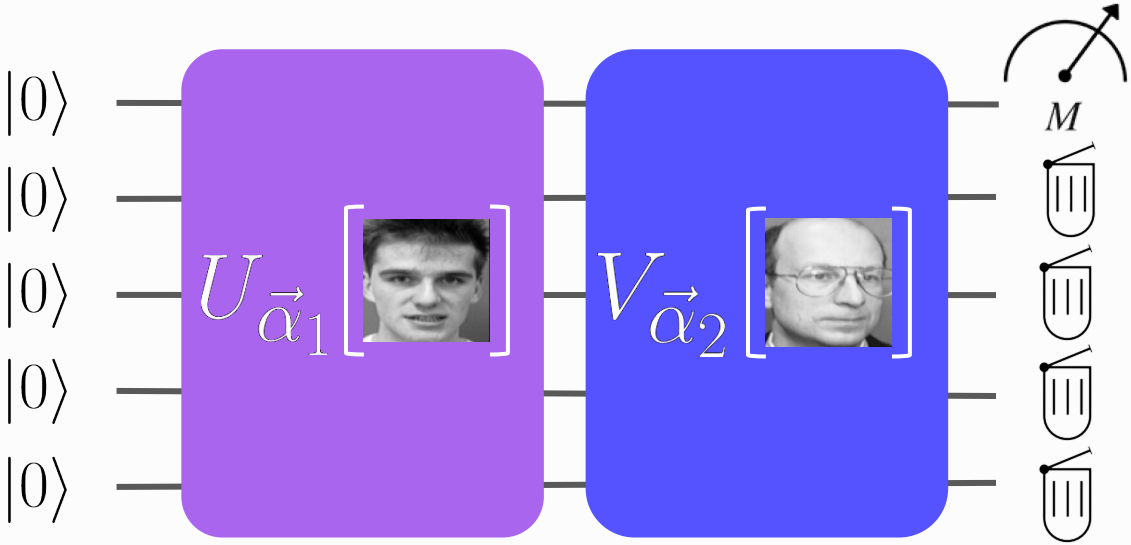


$$f(\vec{x}_i, \vec{x}_j, \vec{\theta}) \approx y_{ij}$$

y_{ij}

PyTorch

Quantum Similarity Network



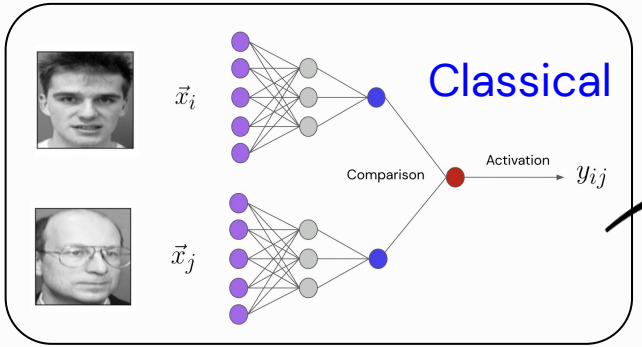
$$f(\vec{x}_i, \vec{x}_j, \vec{\theta}) \approx y_{ij}$$

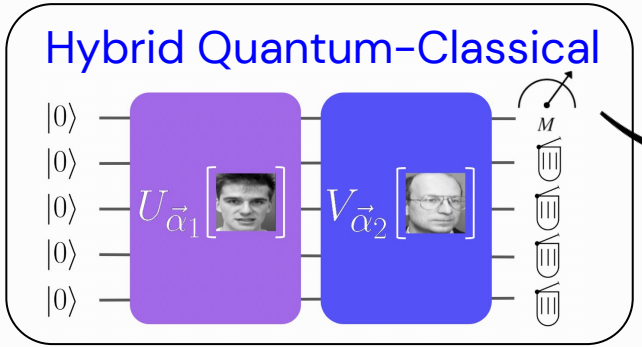
Can be as simple as the probability of measuring a single qubit in the 1 or 0 state.

Radha, S.K. and Jao, C., 2022. *Generalized quantum similarity learning.* *arXiv preprint* *arXiv:2201.02310*.

PENNYLANE

Enter Covalent





Covalent

Classical HPC

aws

slurm
workload manager

Quantum-classical compute

Amazon Braket

Qiskit

IBM

85674602...-4ff9e320957f

Completed | 86 / 86

Overview

Started - Ended

Sep 16, 15:03:40 - Sep 16, 15:04:59

Runtime

1m

Directory

/Users/jbaker/C...c2-b3e9-4ff9e320957f

Input

```
{"args": [], "kwargs": {"train_path": "/Users/jb
```

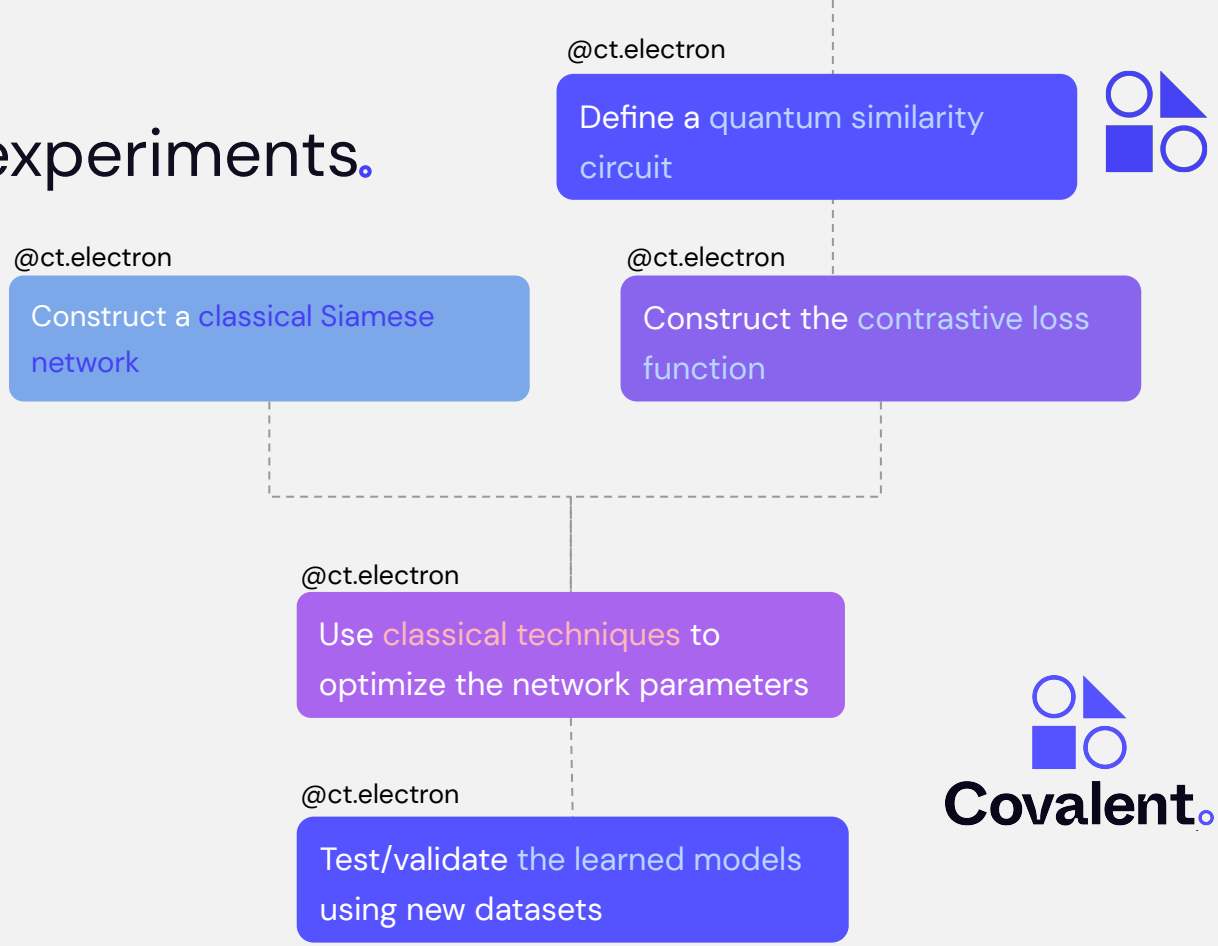
Result

```
(80.0, 82.0, SiameseNetwork(\n  (cnn1): Sequenti
```

Executor: **dask**

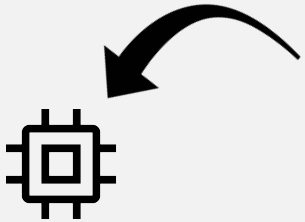
```
@ct.lattice
def quantum_classical_workflow(train_path, test

    # Run the classical sublattice
    results_classical, loss_history_classical,
        classical_workflow(train_path=pixel_tr
                           test_path=pixel_tes
                           train_batch=train_b
                           pytorch_optimizer=p
                           epochs=epochs,
                           lr=lr_c,
                           print_intermediate=
```

Modularize your experiments.



Covalent.

Modularize your experiments.



Can dispatch specific @ct.electrons
to *different* remote devices

@ct.electron

Define a quantum similarity circuit

@ct.electron

Construct a classical Siamese network

@ct.electron

Construct the contrastive loss function

@ct.electron

Use classical techniques to optimize the network parameters

@ct.electron

Test/validate the learned models using new datasets

Covalent.

Modularize your experiments.

@ct.electron

Define a quantum similarity circuit

@ct.electron

Construct a classical Siamese network

@ct.electron

Construct the contrastive loss function

Can dispatch specific @ct.electrons to *different* remote devices

@ct.electron

Use classical techniques to optimize the network parameters

@ct.lattice

Call all electrons in the desired order to create a **heterogeneous** workflow

@ct.electron

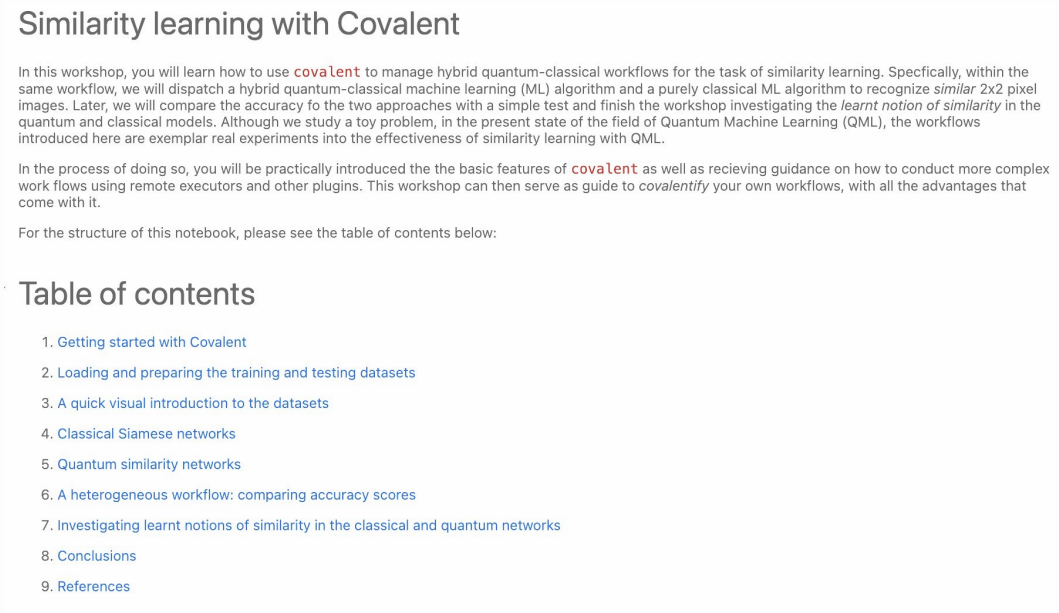
Test/validate the learned models using new datasets

Covalent.

The workshop

- 1 hour workshop.
- Use classical and quantum methods for similarity learning and compare them.
- Let us know about your own workflows!

similarity_learning.ipynb



Similarity learning with Covalent

In this workshop, you will learn how to use `covalent` to manage hybrid quantum-classical workflows for the task of similarity learning. Specfically, within the same workflow, we will dispatch a hybrid quantum-classical machine learning (ML) algorithm and a purely classical ML algorithm to recognize *similar* 2x2 pixel images. Later, we will compare the accuracy fo the two approaches with a simple test and finish the workshop investigating the *learnt notion of similarity* in the quantum and classical models. Although we study a toy problem, in the present state of the field of Quantum Machine Learning (QML), the workflows introduced here are exemplar real experiments into the effectiveness of similarity learning with QML.

In the process of doing so, you will be practically introduced the the basic features of `covalent` as well as recieving guidance on how to conduct more complex work flows using remote executors and other plugins. This workshop can then serve as guide to *covalentify* your own workflows, with all the advantages that come with it.

For the structure of this notebook, please see the table of contents below:

Table of contents

1. Getting started with Covalent
2. Loading and preparing the training and testing datasets
3. A quick visual introduction to the datasets
4. Classical Siamese networks
5. Quantum similarity networks
6. A heterogeneous workflow: comparing accuracy scores
7. Investigating learnt notions of similarity in the classical and quantum networks
8. Conclusions
9. References

SETUP

- Clone the Github repository

```
git clone https://github.com/AgnostiqHQ/tutorials_covalent_ieee_2022.git
```

- Download Miniconda
- Install the conda environment (`environment.yml` in root directory)

```
conda env create -f environment.yml
```

- Activate environment

```
conda activate ieee_covalent
```

- Add to kernel list

```
python -m ipykernel install --user --name=ieee_covalent
```