KubeCon | CloudNativeCon

North America 2018

# Running Serverless HPC Workloads on Top of Kubernetes and Jupyter Notebooks

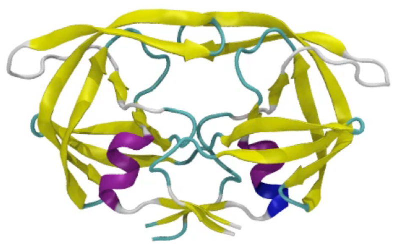
Christopher Woods, University of Bristol, UK

EPSRC
Engineering and Physical Sciences Research Council

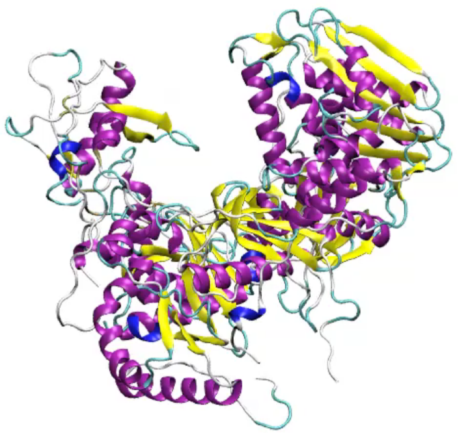
University of Bristol

https://chryswoods.com/talks

# Research at the Cutting Edge



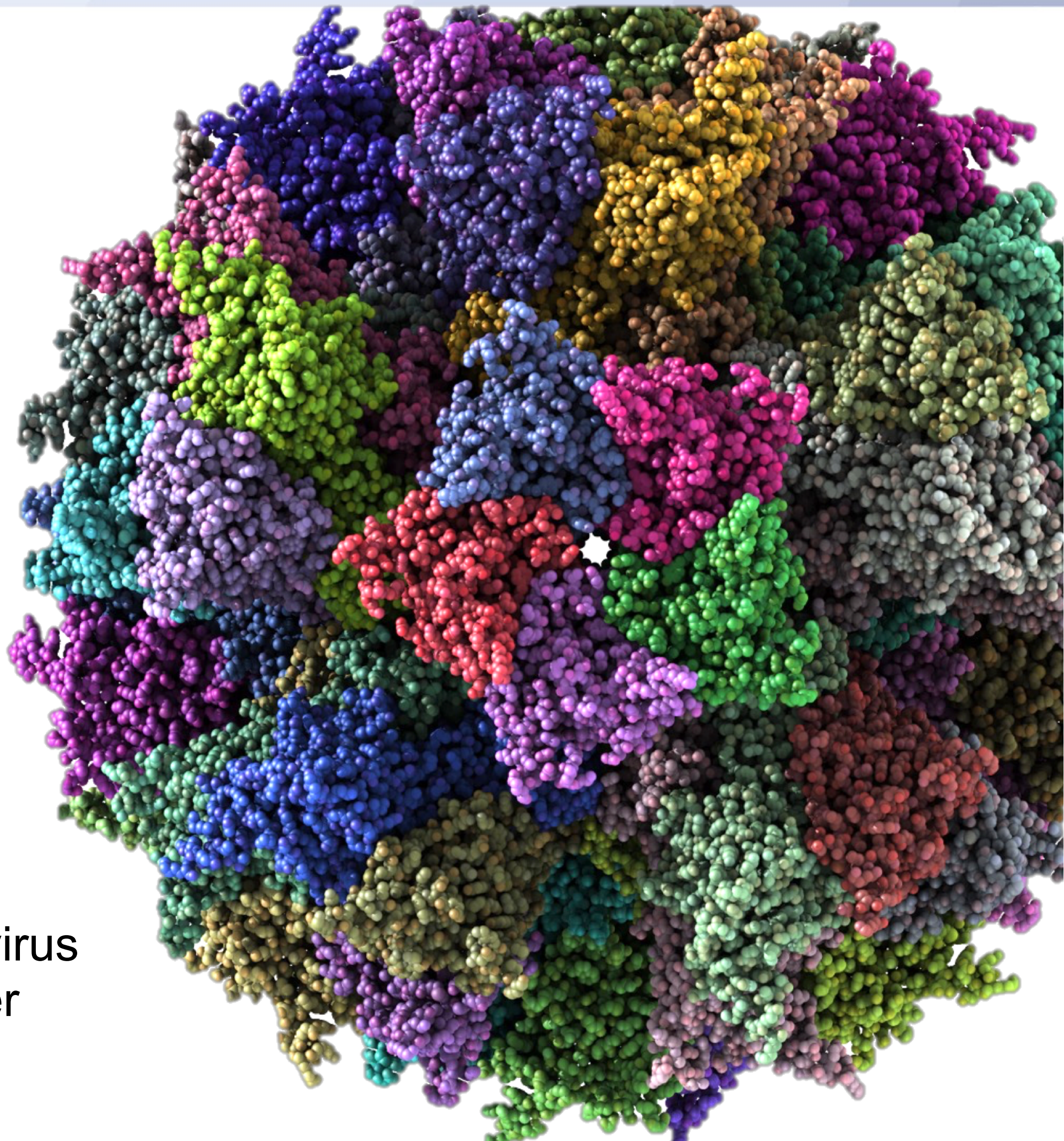
HIV Protease



HIV Reverse Transcriptase

Increasing size and complexity of simulations

ADDomer pseudo-virus immuno-promoter



# High Performance Computing




1. SSH to login node

2. Upload input (rsync)

3. Submit job to Q

4. Wait in Q

5. Wait for job

6. Analyse results

5. Download output (rsync)

# Interactive molecular dynamics

BioSimSpace is a great tool for playing around with molecular simulations directly and interacting with them in real-time. In this notebook you'll learn how to use BioSimSpace to set up and run an equilibration protocol, then query the running process for information, plot graphs of the latest data, visualise molecular configurations, and analyse trajectory data.

Before we get started, let's import BioSimSpace so that it's available inside of our notebook.

In [ ]:
```
import BioSimSpace as BSS
```

## Creating a molecular system

First of all we need to load a molecular system.

In [ ]:
```
system = BSS.IO.readMolecules(["amber/ala/ala.crd", "amber/ala/ala.top"])
```

We have now created a molecular system. The system consists of an alanine dipeptide molecule in a box of water. To show the number of molecules in the system, run:

In [ ]:
```
system.nMolecules()
```

## Defining a simulation protocol

BioSimSpace provides functionality for defining various simulation protocols. In this notebook we will construct a typical simulation workflow that uses a sequence of simple protocols, with the output of one forming the input of the next:

1. *Minimisation:* Energy minimisation the molecular system.
2. *Equilibration:* Equilibration of the system to a target temperature.
3. *Production:* Regular molecular dynamics, run at fixed temperature.
4. *Custom*: A user defined protocol, e.g. a config file for a molecular dynamics package.

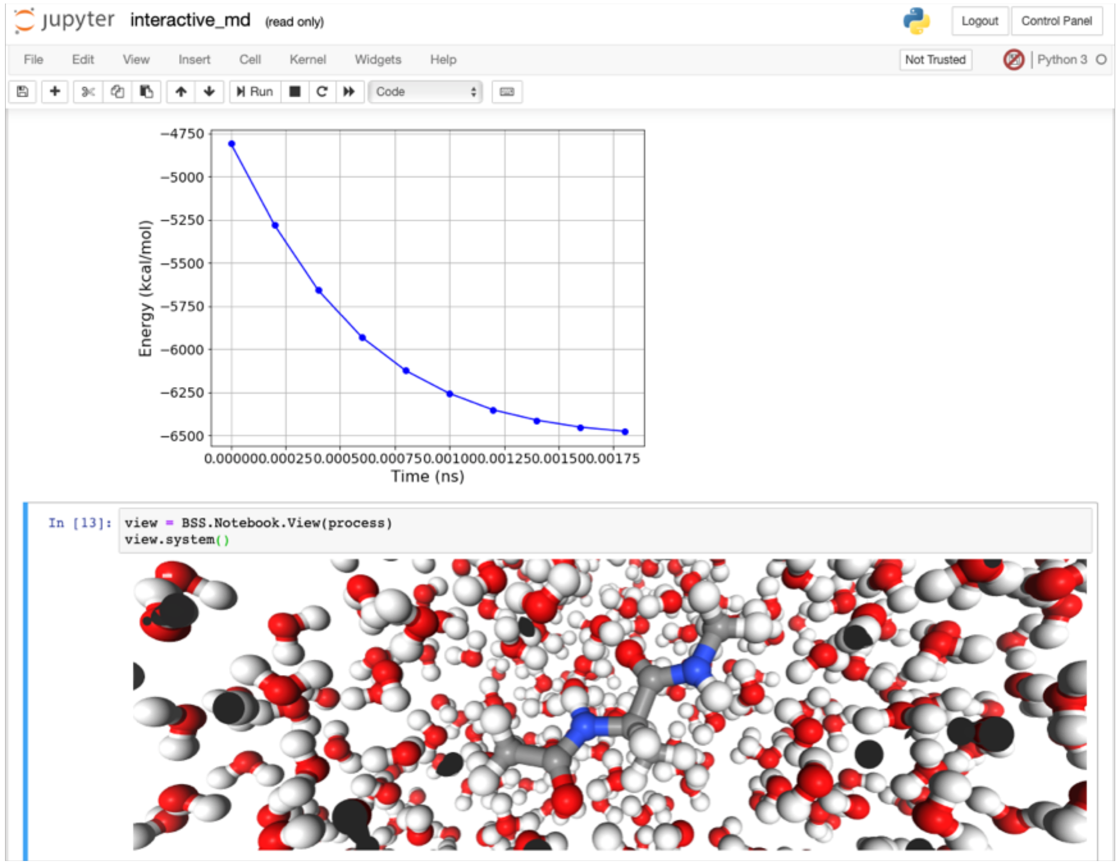
When defining a protocol we are configuring the type of simulation that we wish to run, as well as any options for the particular simulation. For example, to create a default equilibration protocol:

```
protocol = BSS.Protcol.Equilibration()
```

This defines a 0.2 nanosecond equilibration protocol at a temperature of 300 Kelvin. For convenience, let's reduce the runtime. We'll also perform a heating protocol and will restrain the position of atoms in the backbone.

# Jupyter Notebooks






https://jupyter.org – https://biosimspace.org

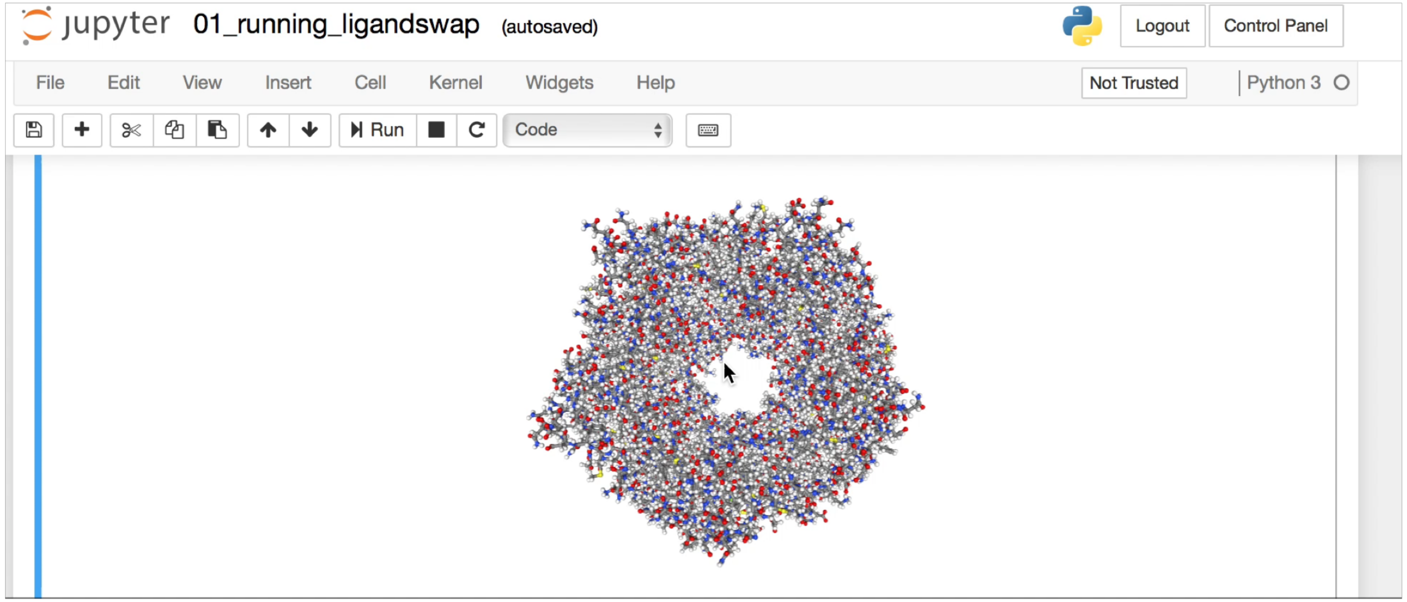
- **Jupyter notebook combines the**;
  - description of the experiment
  - code to run the experiment
  - code to analyse the results
  - graphs and 3D visualisations of the results
  - conclusions of the experiment
- **They contain everything needed to describe and reproduce the experiment**

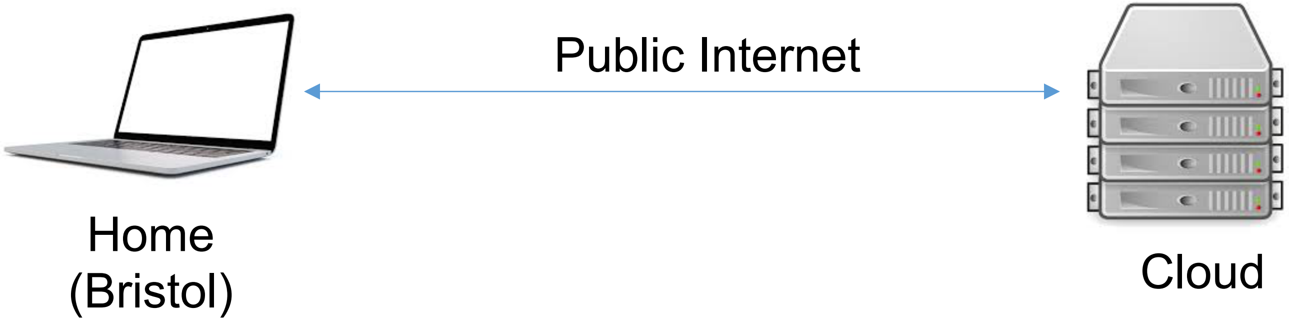
# Notebooks are interactive papers



- **All compute and data sits in the cloud. Only the information needed to render the data in the notebook is transmitted over the network**
- **Hugely useful for open and reproducible science**
- **Notebooks are, in effect, interactive scientific papers ☺**

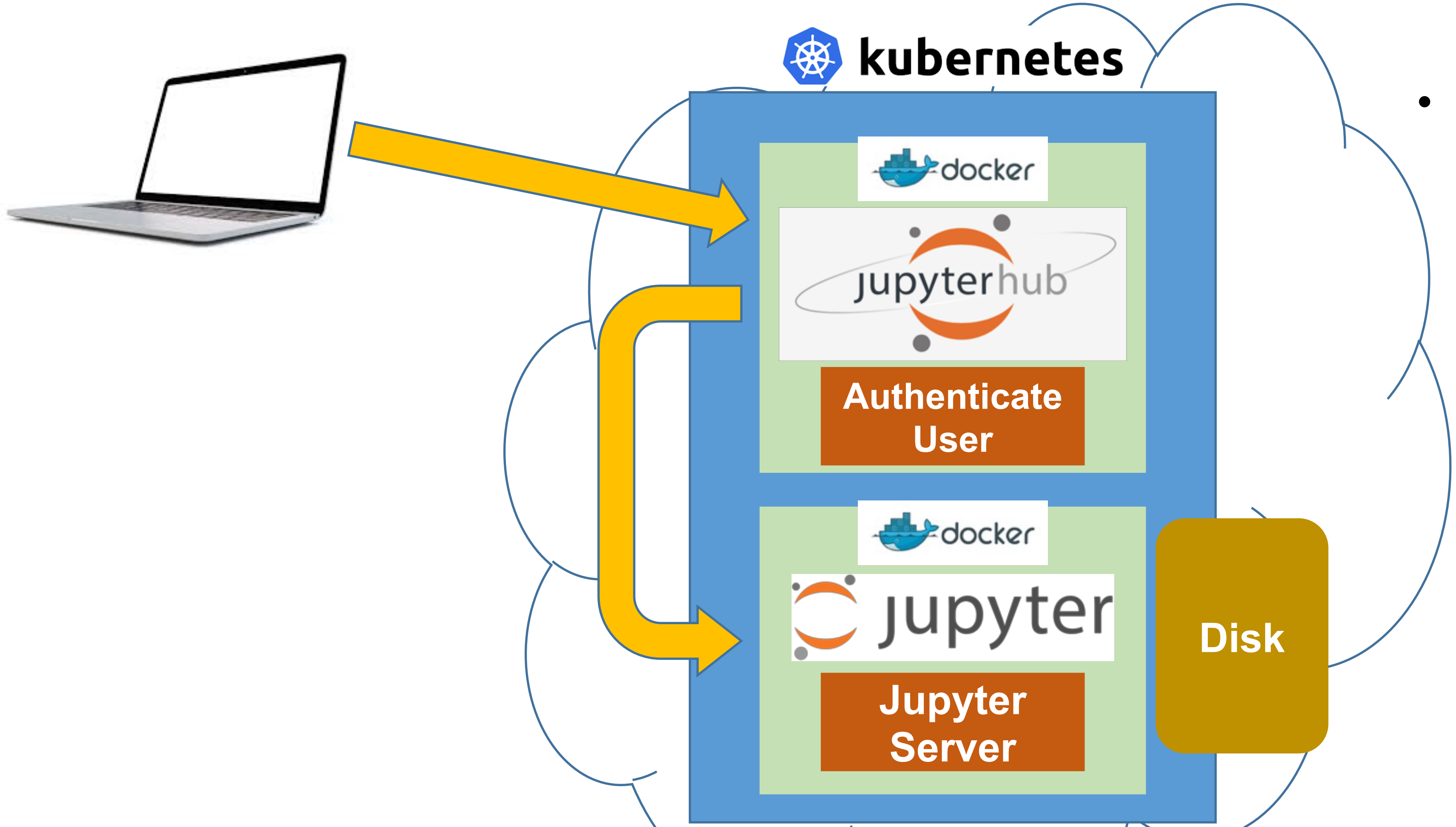
**…but where is the compute to run them? How can anyone reproduce the results if they don't have a local HPC machine?**



# Kubernetes and JupyterHub






- **JupyterHub running on k8s**
  - Easy to use helm chart!
  - Great community and instructions
  - Works with lots (all?) cloud kubernetes services, or roll-your-own clusters

Google Kubernetes Engine
Microsoft Azure Kubernetes Service
Oracle Cloud Infrastructure Container Engine for Kubernetes

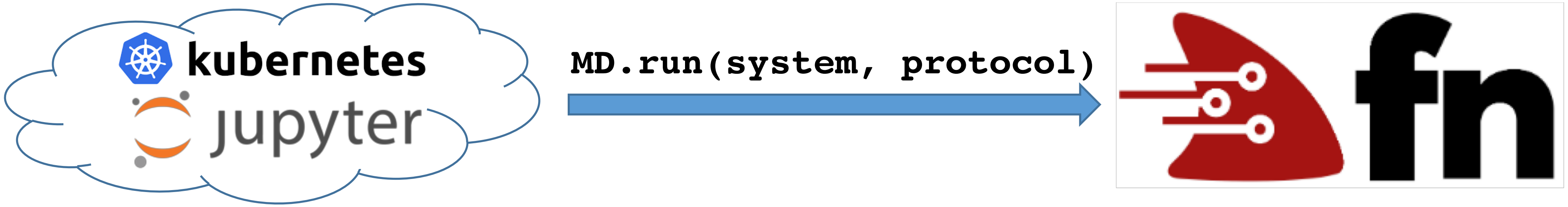
https://zero-to-jupyterhub.readthedocs.io/en/stable/

# Simulations as Serverless Functions

We now have everything that is needed to create a process object. To do so, run:

```
In [5]: process = BSS.MD.run(system, protocol)

In [6]: process.isRunning()

Out[6]: True
```

- The above line starts and runs a molecular dynamics simulation
- However, we cannot run this in the k8s pod, as the hardware is too tiny…
  - (…or else the k8s cluster would be too expensive)
- Instead we burst out to HPC hardware using a “serverless” function service



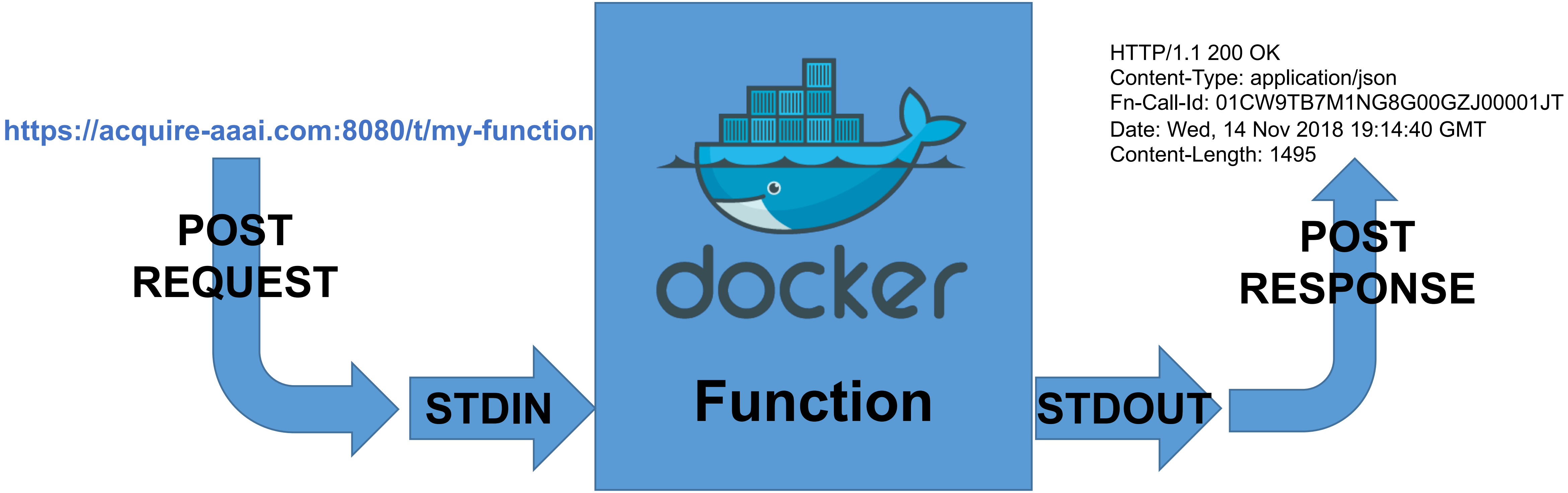

**Auto-scaling 1 or 2 core VMs**

**Fn service running on 52-core HPC nodes**

# Fn Serverless : https://fnproject.io



**Fn is an event-driven, open source, Functions-as-a-Service (FaaS) compute platform that you can run anywhere.**




Code to run the function is wrapped into a docker container. This is allocated to hardware in response to a trigger (e.g. https). Input data is encoded via POST and piped in as STDIN to the container. This is processed by the function, with resulting STDOUT returned as a HTTP response

# Fn Serverless : https://fnproject.io



- Function is **ANY** code (and associated software) that can be packaged into a docker container
- HTTP request is piped in as standard input
- Anything written to standard output is returned as the HTTP response
- Anything written to standard error is logged
- Functions can be synchronous (respond immediately) or asynchronous
- Asynchronous functions return a CALL_ID that can be queried to get progress, cancel function or collect output, thereby supporting long-running functions
- **Supports *ANY* language! Development kits for Go, Python, Java, Ruby, Node, and Rust simplify function writing and automate creation of docker containers**

# Fn Serverless : https://fnproject.io



**Fn FDKs make it easy to write functions that use JSON as the input/output format, and that can “stay hot”**

https://acquire-aaai.com:8080/t/my-function

**POST REQUEST name=“Chris”**

**JSON**

{“name”: ”chris”}

```
import fdk
import json


async def handler(ctx, data=None):
    name = "World"
    if data and len(data) > 0:
        body = json.loads(data)
        name = body.get("name")
    return "Hello {0}".format(name)


if __name__ == "__main__":
    fdk.handle(handler)
```

Python function wrapped via Fn Python Devkit (FDK) handling JSON input and writing JSON output

**JSON**

“Hello Chris”

**POST RESPONSE**

“Hello Chris”

# Simulations as Fn Functions

```
In [ ]: import BioSimSpace as BSS

In [ ]: system = BSS.IO.readMolecules(["amber/ala/ala.crd", "amber/ala/ala.top"])

In [ ]: # Initialise a short equilibration protocol.
        protocol = BSS.Protocol.Equilibration(runtime=0.05*BSS.Units.Time.nanosecond,
                                              temperature_start=0*BSS.Units.Temperature.kelvin,
                                              temperature_end=300*BSS.Units.Temperature.kelvin,
                                              restrain_backbone=True)

In [ ]: process = BSS.MD.run(system, protocol)

In [ ]: # Generate a plot of time vs temperature.
        plot1 = BSS.Notebook.plot(process.getTime(time_series=True),
            process.getTemperature(time_series=True))

        # Generate a plot of time vs energy.
        plot2 = BSS.Notebook.plot(process.getTime(time_series=True),
            process.getTotalEnergy(time_series=True))

In [ ]: view = BSS.Notebook.View(process)
        view.system()
```

kubernetes

jupyter

**Auto-scaling 1 or 2 core VMs**

fn

**Fn service running on 52-core HPC nodes**

# Simulations as Fn Functions

```
In [1]: import BioSimSpace as BSS

In [2]: system = BSS.IO.readMolecules(["amber/ala/ala.crd", "amber/ala/ala.top"])

In [3]: # Initialise a short equilibration protocol.
        protocol = BSS.Protocol.Equilibration(runtime=0.05*BSS.Units.Time.nanosecond,
                                              temperature_start=0*BSS.Units.Temperature.kelvin,
                                              temperature_end=300*BSS.Units.Temperature.kelvin

In [4]: process = BSS.MD.run(system, protocol)

        # Generate a plot of time vs energy.
        plot2 = BSS.Notebook.plot(process.getTime(time_series=True),
            process.getTotalEnergy(time_series=True))

In [ ]: view = BSS.Notebook.View(process)
        view.system()
```

async call

MD.run(…)

kubernetes

jupyter

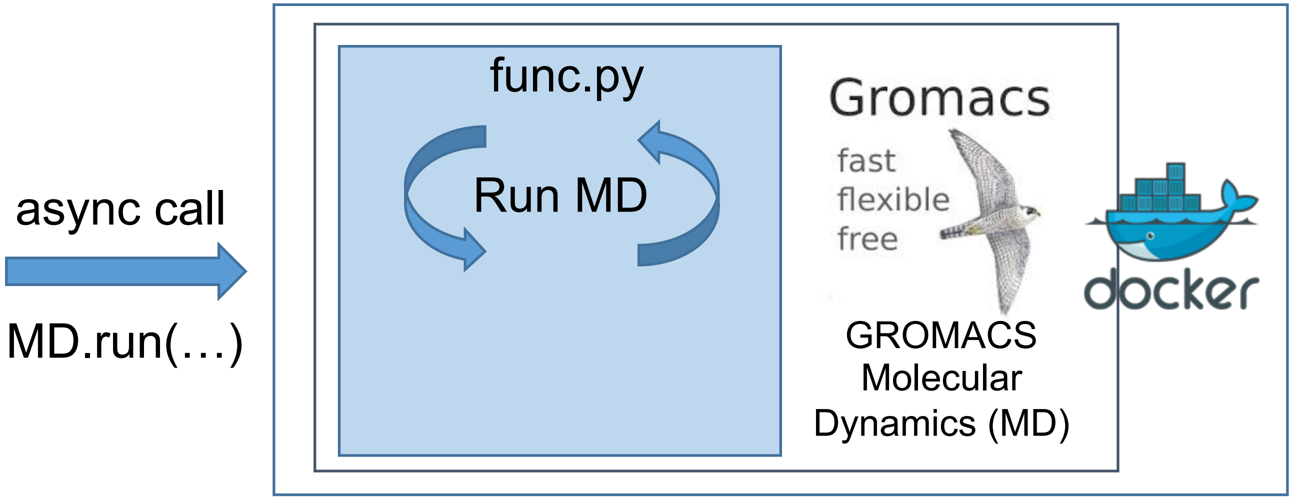
fn

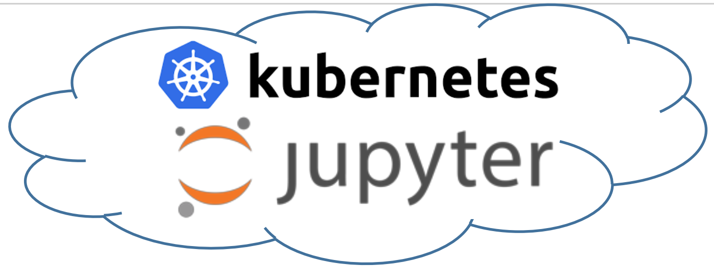
**Auto-scaling 1 or 2 core VMs**

**Fn service running on 52-core HPC nodes**

# Simulations as Fn Functions

```
In [1]: import BioSimSpace as BSS

In [2]: system = BSS.IO.readMolecules(["amber/ala/ala.crd", "amber/ala/ala.top"])

In [3]: # Initialise a short equilibration protocol.
        protocol = BSS.Protocol.Equilibration(runtime=0.05*BSS.Units.Time.nanosecond,
                                              temperature_start=0*BSS.Units.Temperature.kelvin,

In [4]: process = BSS.MD.run(system, protocol)

        # Generate a plot of time vs energy.
        plot2 = BSS.Notebook.plot(process.getTime(time_series=True),
            process.getTotalEnergy(time_series=True))

In [ ]: view = BSS.Notebook.View(process)
        view.system()
```



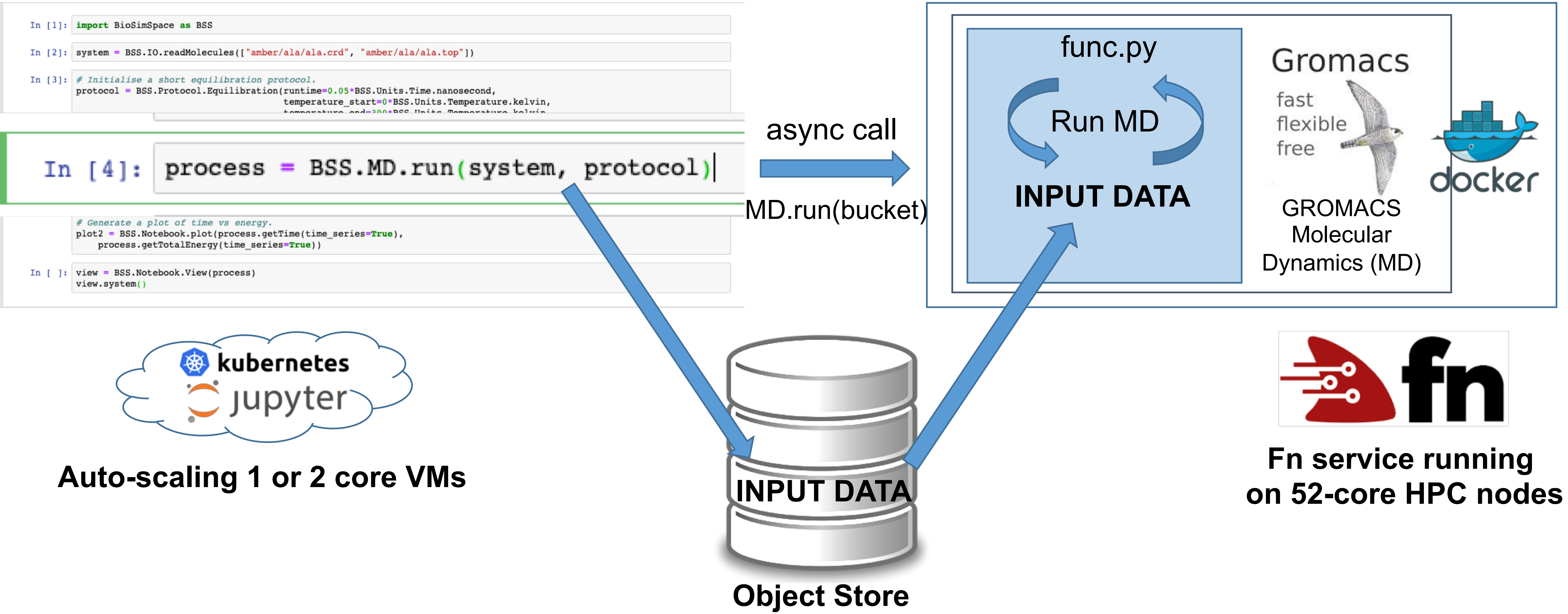




**Auto-scaling 1 or 2 core VMs**

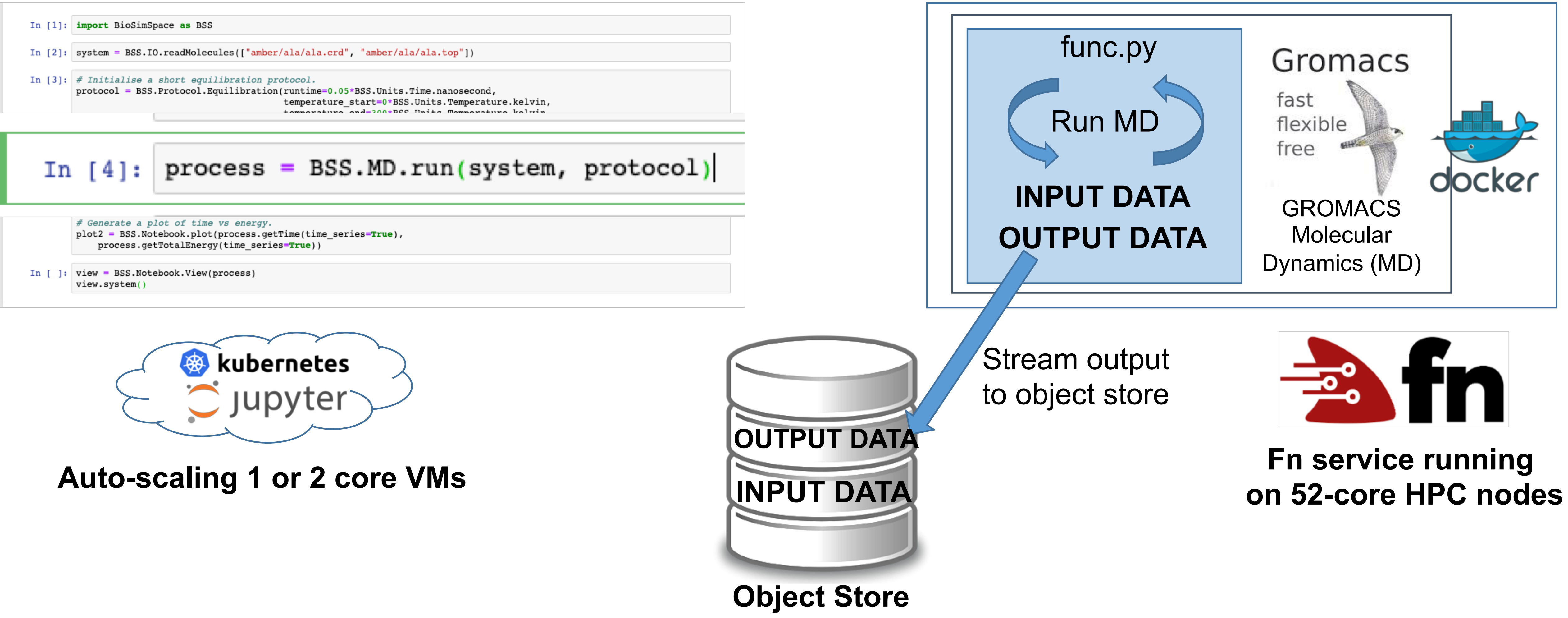
**Fn service running on 52-core HPC nodes**

# Serverless + Object Store ☺

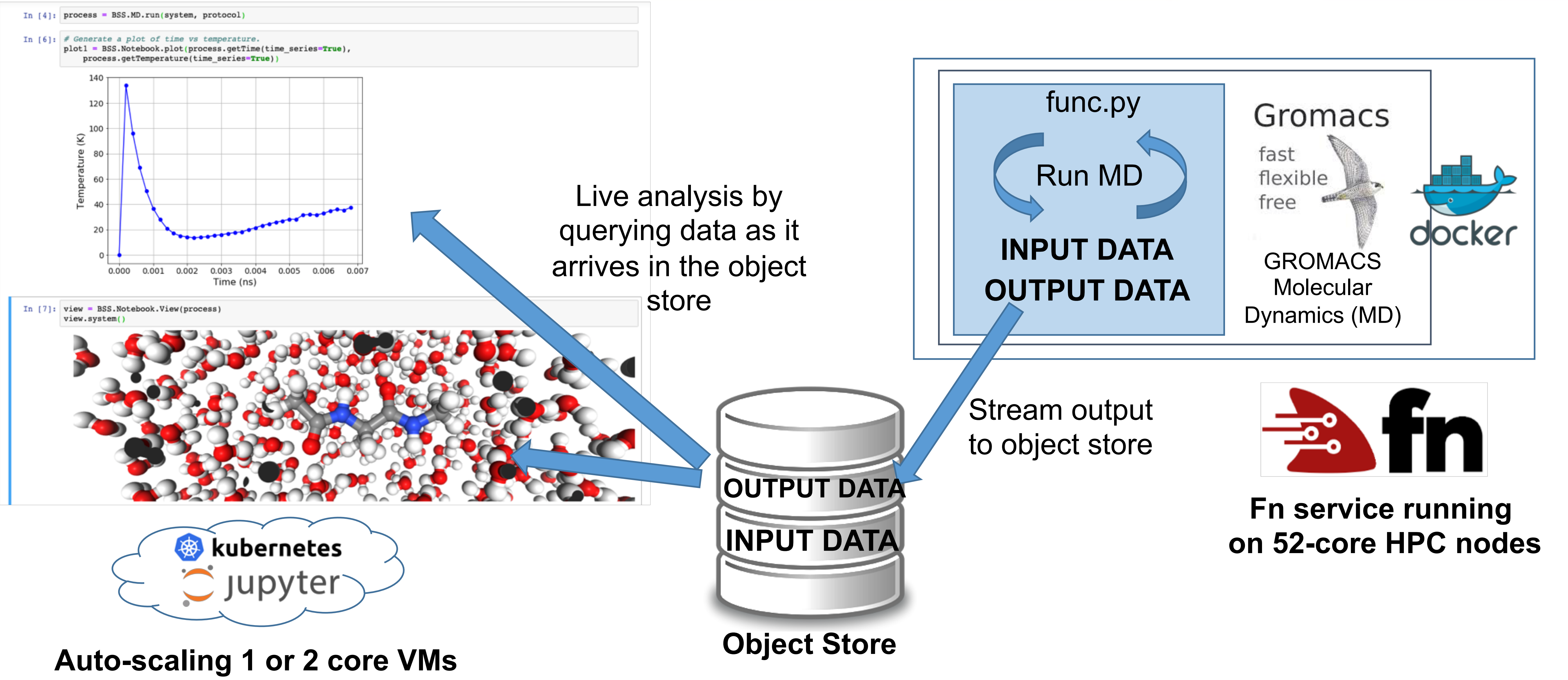




# Serverless + Object Store ☺



# Serverless + Object Store ☺



# Simulations as Fn Functions



- Different “molecular dynamics” Fn function calls can be associated with different hardware
  - Enables high memory, big CPU or GPU nodes to be allocated on demand in response to function calls
- The Fn framework is open source and cross-platform, so can work on any cloud
- Works with any application that can be packaged into a docker container
- Object Store used as intermediary to keep messages small. Benefit is output data can be assigned a unique URL / DOI and immediately published
- **Simple framework that allows ANYONE to run HPC simulations by calling the Fn function via a public URL**

# Anyone can run simulations…!



- **Simple framework that allows ANYONE to run HPC simulations by calling the Fn function via a public URL!**

**That could get expensive…!**

**Looks like we need some user authentication, access control and accounting…**

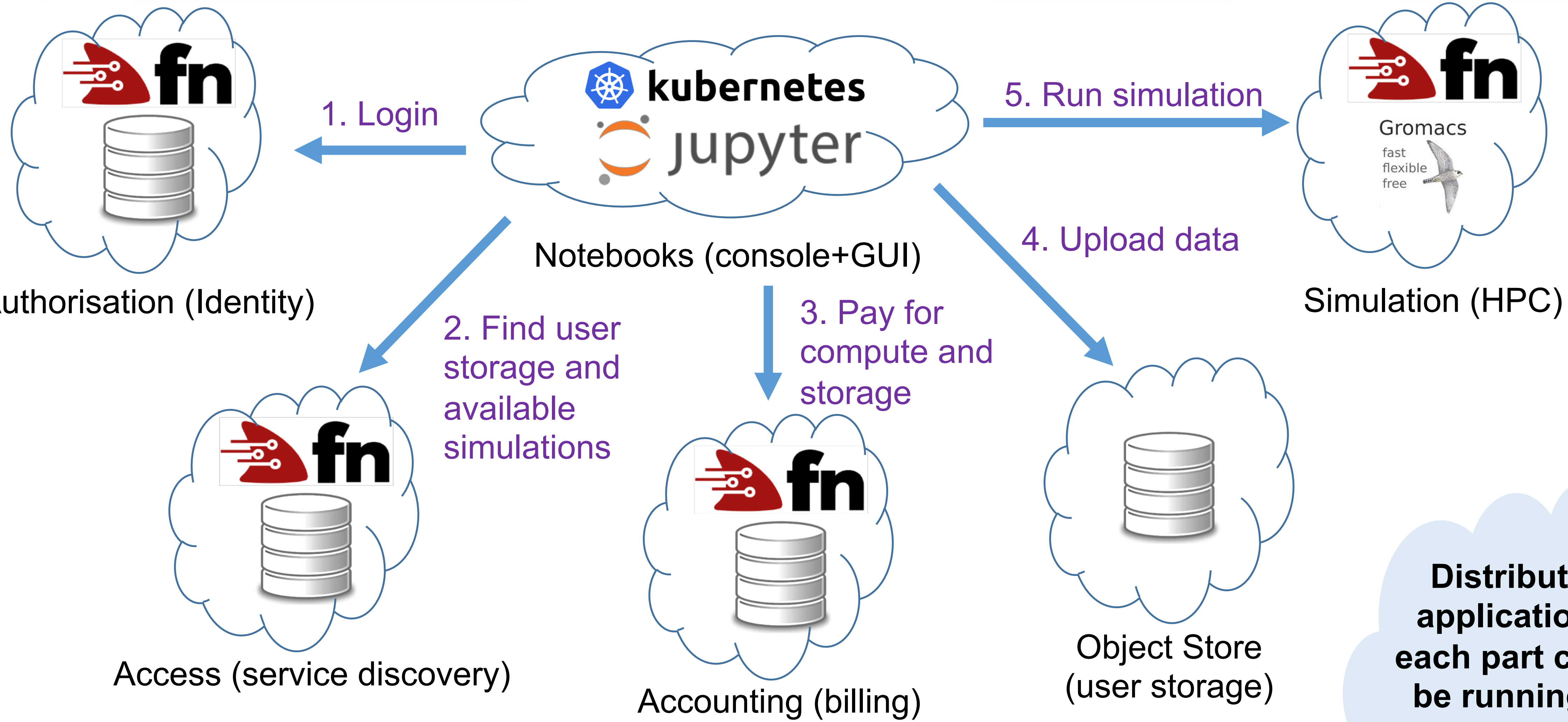
# Authorisation (Identity)

```
In [ ]: from Acquire.Client import User

In [ ]: user = User("chryswoods")

In [ ]: (url,qrcode) = user.request_login()

In [ ]: qrcode

In [ ]: from IPython.core.display import HTML
        print(url)
        HTML("<a href='%s'>Login here</a>" % url)

In [ ]: user.wait_for_login()

In [ ]: user.is_logged_in()

In [ ]: user.logout()

In [ ]:
```

- Built an authorisation (identity) service on top of Fn serverless and object store for state
- “request_login” call from the notebook calls “request_login” serverless function. This looks up user details from object store and returns a unique login URL
- Login page also served as html from an Fn function
- Notebook can wait for the login to complete, and uses security tokens to authenticate with simulation function service

# Authorisation, Access, Accounting






**Distributed application – each part could be running in different clouds!**

# Serverless solves everything?




Modified from https://www.flickr.com/photos/-marlith-/2634898429

# Cold Start is REALLY painful!



```
t1 = datetime.datetime.now()
response = Acquire.Service.call_function(function_url)
t2 = datetime.datetime.now()
print( "Call took {0} s".format((t2-t1).total_seconds()))
```

```
Call took 4.087868 s
```

```
t1 = datetime.datetime.now()
response = Acquire.Service.call_function(function_url)
t2 = datetime.datetime.now()
print( "Call took {0} s".format((t2-t1).total_seconds()))
```

```
Call took 0.092248 s
```

**In 4087ms I expect my HPC code to perform ~40B floating point calculations and simulate ~5000 steps of protein dynamics!**

**Spending >4s just to call a single function is embarrassing!**

# Cold Start is REALLY painful!



```
t1 = datetime.datetime.now()
response = Acquire.Service.call_function(function_url)
t2 = datetime.datetime.now()
print( "Call took {0} s".format((t2-t1).total_seconds()))
```

```
Call took 4.087868 s
```

```
t1 = datetime.datetime.now()
response = Acquire.Service.call_function(function_url)
t2 = datetime.datetime.now()
print( "Call took {0} s".format((t2-t1).total_seconds()))
```

```
Call took 0.092248 s
```

**In 4087ms I expect my HPC code to perform ~40B floating point calculations and simulate ~5000 steps of protein dynamics!**

**Spending >4s just to call a single function is embarrassing!**

- **Cold-start of a function is SLOW**
  - Container has to be allocated
  - Python interpreter needs to start
  - Modules must be imported
  - Script must run
  - State must be reloaded if needed
- **Once called, the function is left running so it is ready to process the next request (it is hot)**
- **Someone has to pay the cost of “heating” the function**

# Packaging sub-functions into apps

https://acqui.red:8080/t/route

**POST REQUEST**
**function="hello"**
**Name="Chris"**

**JSON**

**{"function":"hello", "name":"chris"}**

```
import fdk
import json

async def hello(args):
    name = "World"
    if "name" in args:
        name = args["name"]
    return "Hello {0}".format(name)

async def handler(ctx, data=None):
    if data and len(data) > 0:
        body = json.loads(data)
        func = body["function"]
        if func == "hello":
            return hello(body)
        elif func == "goodbye":
            return goodbye(body)

    return "UNHANDLED FUNCTION"

if __name__ == "__main__":
    fdk.handle(handler)
```

**JSON**

"Hello Chris"

**POST RESPONSE**

"Hello Chris"

# Packaging sub-functions into apps

KubeCon | CloudNativeCon North America 2018

https://acquire-aaai.com:8080/t/identity/route

{"function" : "request_login",
"username" : "chryswoods",
"public_key" : "XXXXXXXX",
"public_certificate" : "XXXXX"}

**route.py**

get_keys
get_status
login
logout
register
**request_login**
setup
root
warm
whois

https://acquire-aaai.com/t/identity/s?id=19b187fc

{"session_uid" : "XXXXXX",
"login_url" : "https://acquire-aaai.com:8080/t/identity/s?id=19b187fc",
"user_uid" : "XXXXXXXX"}

# Packaging sub-functions into apps



- **Packaging all "sub-functions" into a single "function" that represents the application has many advantages:**
  - **Once one of the sub-functions is hot, all sub-functions are hot**
  - As all sub-functions are in the same docker container, pulling this single container to a node gives it access to all sub-functions
  - **Async functions allow a single threads to handle multiple different sub-function calls at the same time**
  - You can cache state between sub-function calls, e.g. security IAM credentials used to access the object store, or reading rarely-changing data from object store (make use of Python cachetools and @cached decorator)
- **Same security (data leaking) issues as keeping the interpreter hot, i.e. you must trust all code. Don't execute arbitrary (user-supplied) code!!!**

# Profile to minimise startup time

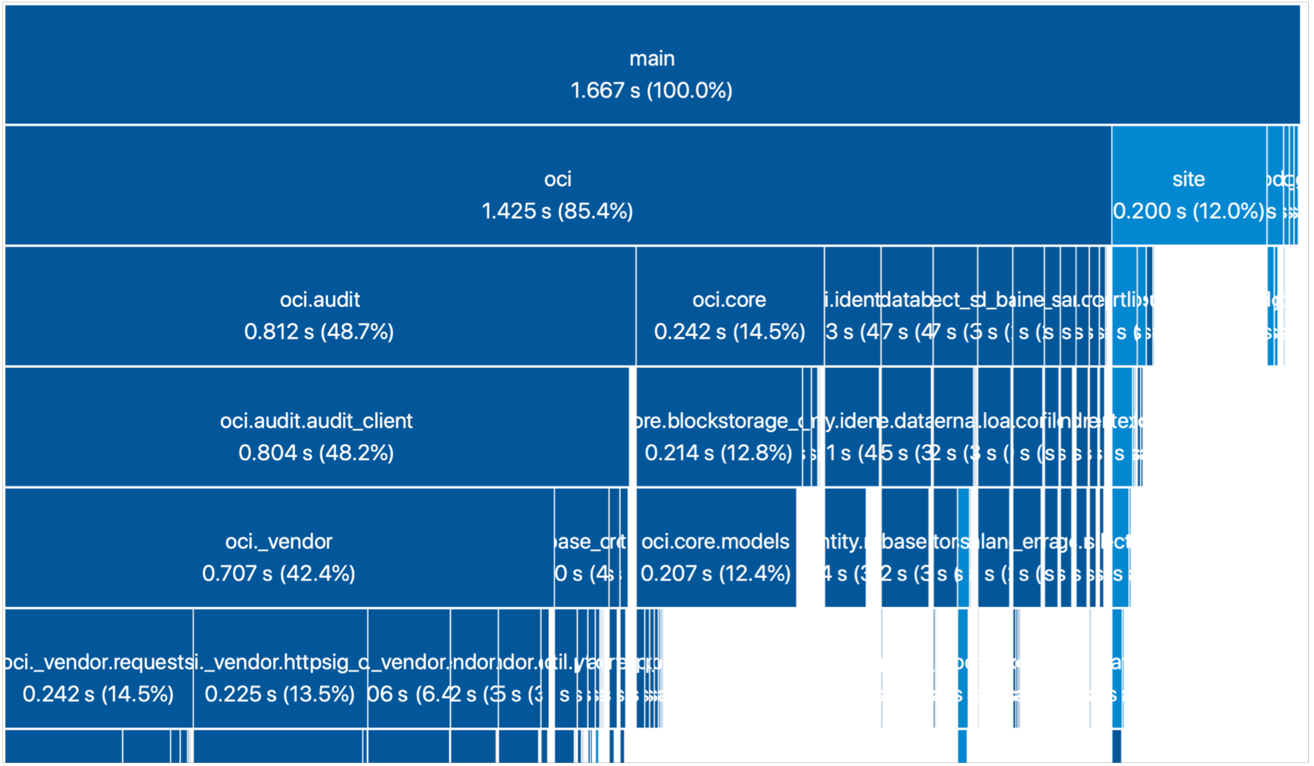


- **Choose a language and runtime that start quickly, e.g. like Python**

```
calculon 18:43:03 ~
:-> time python -c "import json"

real    0m0.061s
user    0m0.023s
sys     0m0.033s
```

# Profile to minimise startup time

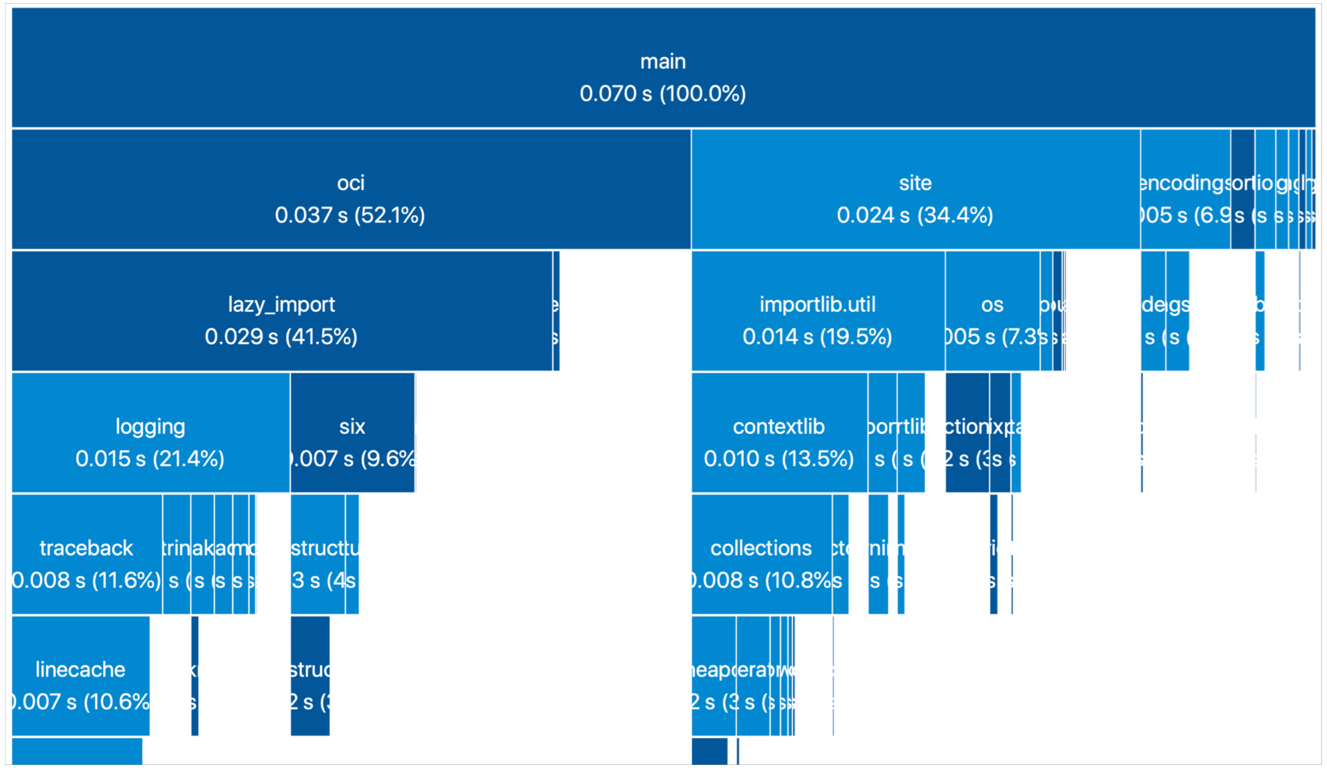
```
calculon 18:50:22 ~
:-> PYTHONPROFILEIMPORTTIME=1 python -c "import oci" 2> profile.txt

calculon 18:50:25 ~
:-> tuna profile.txt
```




- **Choose a language and runtime that start quickly, e.g. like Python**
- **Profile your imports so that you can identify bottlenecks**

# Profile to minimise startup time

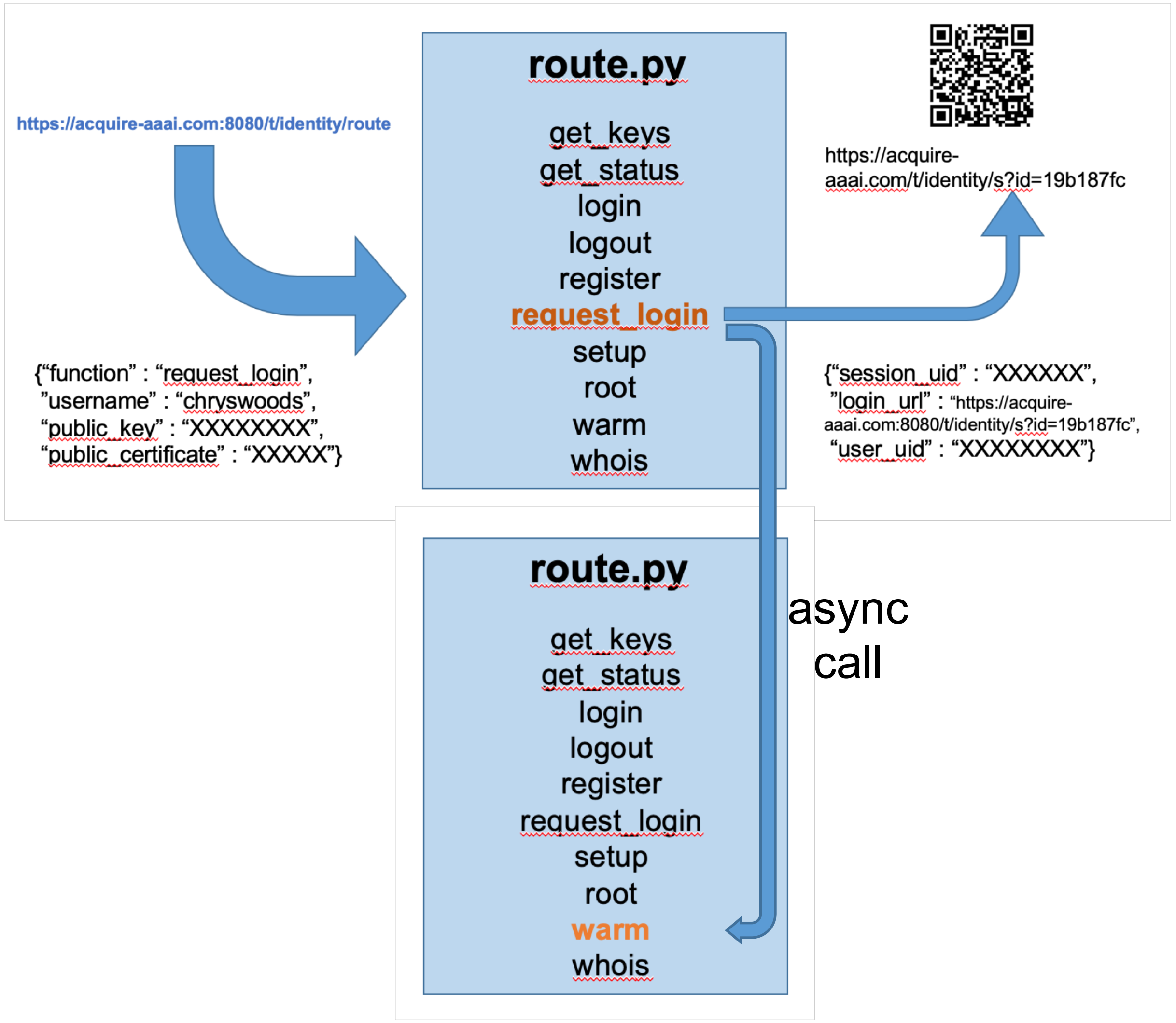
```
calculon 18:50:22 ~
:-> PYTHONPROFILEIMPORTTIME=1 python -c "import oci" 2> profile.txt

calculon 18:50:25 ~
:-> tuna profile.txt
```




- **Choose a language and runtime that start quickly, e.g. like Python**
- **Profile your imports so that you can identify bottlenecks**
- **Use “lazy_import” to delay/remove imports you don’t need or use**

```
import lazy_import as _lazy_import

audit = _lazy_import.lazy_module("oci.audit")
container_engine = _lazy_import.lazy_module("oci.container_engine")
core = _lazy_import.lazy_module("oci.core")
database = _lazy_import.lazy_module("oci.database")
dns = _lazy_import.lazy_module("oci.dns")
email = _lazy_import.lazy_module("oci.email")
file_storage = _lazy_import.lazy_module("oci.file_storage")
identity = _lazy_import.lazy_module("oci.identity")
key_management = _lazy_import.lazy_module("oci.key_management")
load_balancer = _lazy_import.lazy_module("oci.load_balancer")
```

# One hot spare



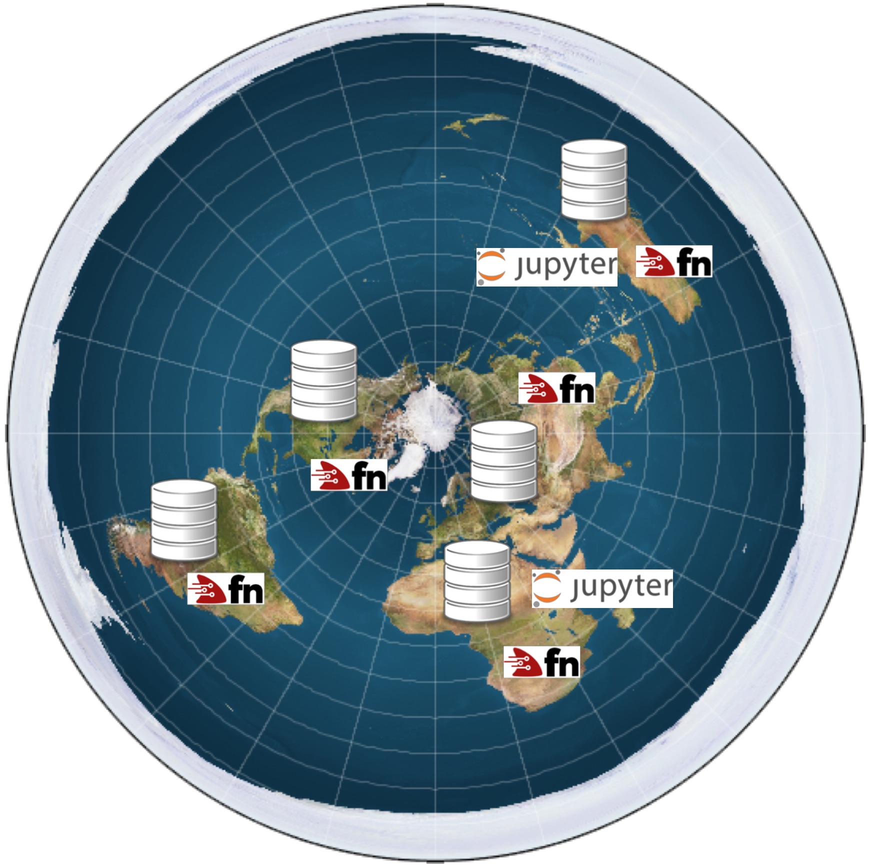

- **Just like in the hardware world, make sure you always have "one hot spare"**
- **Keep one instance of your "route.py" sub-function bundle permanently hot**
  - (Ok, not really serverless, but it's lightly using 1 core, which is pennies per hour… And practicality should always beat idealism)
- **Have route issue an async (non-blocking) function call to "warm". This does nothing except schedule a spare copy of route to be pre-warmed ready for other users**
- **Bundling subfuncs into route means that only one hot spare is needed for the app**

# The Planetary Supercomputer

KubeCon | CloudNativeCon North America 2018

Functions == Processes

Notebooks == Console/GUI




Object Store == Disk/Memory Storage

AAAI == User accounts and resource scheduler

- **Building a service that allows on-demand running of HPC workloads from within interactive Jupyter notebooks with a full user Authentication, Access control and financial Accounting Infrastructure (AAAI)**
- **Fn is an excellent function / serverless platform. Open source ☺**
- **Fully portable – works across clouds!**
- **Notebooks + Serverless + Object Store equals programming the planetary supercomputer**
- **Or, as my students call it, building the Netflix of Simulation**

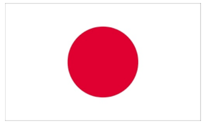
# The Planetary Supercomputer

KubeCon | CloudNativeCon North America 2018

```
In [ ]: from Acquire.Client import User
In [ ]: user = User("chryswoods", identity_url="http://identity-gcp.acquire-aaai.com:8080/t/identi
In [ ]: (url,qrcode) = user.request_login()
In [ ]: qrcode
In [ ]: from IPython.core.display import HTML
        print(url)
        HTML("<a href='%s'>Login here</a>" % url)
In [ ]: user.wait_for_login()
In [ ]: user.is_logged_in()
In [ ]: user.logout()
In [ ]:
In [ ]:
```

```
In [ ]: from Acquire.Client import User
In [ ]: user = User("chryswoods", identity_url="http://identity-azure.acquire-aaai.com:8080/t/iden
In [ ]: (url,qrcode) = user.request_login()
In [ ]: qrcode
In [ ]: from IPython.core.display import HTML
        print(url)
        HTML("<a href='%s'>Login here</a>" % url)
In [ ]: user.wait_for_login()
In [ ]: user.is_logged_in()
In [ ]: user.logout()
In [ ]:
In [ ]:
```

```
In [ ]: from Acquire.Client import User
In [ ]: user = User("chryswoods", identity_url="http://identity.acquire-aaai.com:8080/t/identity")
In [ ]: (url,qrcode) = user.request_login()
In [ ]: qrcode
In [ ]: from IPython.core.display import HTML
        print(url)
        HTML("<a href='%s'>Login here</a>" % url)
In [ ]: user.wait_for_login()
In [ ]: user.is_logged_in()
In [ ]: user.logout()
In [ ]:
In [ ]:
```



Fn running on GCP in Japan

Fn running on Azure in the Netherlands

Fn running on OCI in Germany

Notebooks running in Seattle

OCI Object Store in Germany

# Acknowledgements



My Research Software Engineering (RSE) Group
Andrew Williams, Chris Edsall, Lester Hedges, Matt Williams

BioSimSpace Research Team
Julien Michel, Antonia Mey, Adrian Mulholland, Charlie Laughton, Francesco Gervasio

EPSRC for funding (EP/N018591/1 and EP/P022138/1)

Oracle for providing a lot of compute time and extremely valuable discussions with cloud engineers and the Fn development team. Special thanks to Phil Bates and Gerardo Viedma

Microsoft and Google for providing cloud time to demonstrate the portability of the system

The countless open source developers behind Fn, Jupyter, Kubernetes, Docker, Gromacs, Linux etc. who all contribute their code to a common pool so that we can all build a better world together

**The conference organisers for accepting my talk and you for attending**

https://chryswoods.com/talks


KubeCon
CloudNativeCon
North America 2018