examples for `hostess.aws.ec2`

introduction

`hostess.aws.ec2` is a collection of utilities for working with EC2 instances.

`Instance` and `Cluster` are its centerpiece classes. They are abstractions for, respectively, single EC2 instances and groups of EC2 instances. By offering both managed interaction with the EC2 API and a rich set of remote procedure call (RPC) capabilities, they attempt to make distributed workflows as conceptually simple and immediate as local ones.

requirements

1. You need appropriate AWS permissions to perform any actions that call the EC2 API. You cannot, for example, use `ls_instances()` without the ListInstances permission, or `Instance.start()` without the StartInstances permission for the particular instance you are attempting to start. (A complete discussion of AWS permissions management is beyond the scope of this document.)

   By default, `hostess` uses the 'default' profile from ~/.aws/credentials. This can be modified in `hostess.config.user_config` or by manually constructing a client with `hostess.aws.utilities.init_client` and passing it as the `client` kwarg to an `Instance` or `Cluster` constructor.

   If you are in a situation where you have SSH access to, but not AWS permissions for, an EC2 instance you'd like to make `hostess` RPCs on, you should ignore the fact that it is an EC2 instance and simply use `hostess.ssh.SSH`, which underlies the RPC capabilities of this module.

2. The RPC functionality offered by this module relies on SSH, so you need SSH access to an instance to make RPCs on it. Specifically:

   A. The inbound traffic rules of the instance's security group must permit SSH access from your IP. (If you create a security group with `hostess`, it will set this up automatically.)
   B. `hostess` supports only keyfile-based authentication, so you must have the correct keyfile for the instance, and `hostess` must be able to find it. If it shares a filename with the key name known to AWS and is in your `~/.ssh` folder, `hostess` will find it automatically. If this is not the case, you can manually specify a path to the keyfile when constructing an `Instance` or `Cluster`. You can also change the default search paths in `hostess.config.user_config`. (If you create a security group with

`hostess`, it will also generate a compatible keyfile and save it into your `~/.ssh` folder.)

C. The instance must be running `sshd` and configured to accept incoming connections. You usually will not have to do anything special to set this up: the default configuration of most stock AMIs, including the Ubuntu and Amazon Linux images, is suitable.

Note: If you want to create remote workflows without relying on SSH connections, we recommend looking at the `hostess.station` framework.

caveat and warning

This tutorial will walk you through the creation of several EC2 instances and associated EBS volumes. This will incur costs attached to your AWS credentials. If this tutorial is run without modification, the total costs should be very small (approximately 13 cents). However, if this is unacceptable to you or makes you uncomfortable, you should proceed with caution or not at all. We (the authors of `hostess`) disclaim responsibility. (See the appendix for a breakdown of estimated costs.)

relationship to `boto3`

`aws.ec2.Instance` is designed to be easier to use, more Pythonic, and more powerful than `boto3`'s `Instance` abstraction. However, it does not implement high-level interfaces for the entirety of the EC2 API. Because it works partly by wrapping portions of `boto3`, however, every `aws.ec2.Instance` also grants access to a `boto3` `Instance` object with a shared AWS configuration. If you need access to other parts of the EC2 API inside a `hostess` workflow, you can access this object via the `aws.ec2.Instance.instance_` attribute.

1. launching instances

`Instance` can be used both to work with existing instances and launch new ones. In this section, we'll launch an instance to work with in subsequent sections. We should discuss some preliminaries first.

IMPORTANT: `hostess` performs no automated instance lifecycle management due to its potential to cause process disruption and unintended data loss. If you want to stop or terminate an instance, you must do so explicitly, using methods of the `Instance` object, other interfaces to the EC2 API, shutdown commands on the instance itself, or the EC2 web console. You may find it helpful or illuminating to have the EC2 web console open in another browser tab while for reference while you work through this notebook.

1.1. instance configuration

launch templates

Although it's not mandatory, the easiest and cleanest way to launch instances with `hostess` is to use an existing launch template and give `Instance.launch()` the name of the template. (On the backend, this is because the API requires a launch template to launch an instance, and reusing the same launch template helps ensure consistent behavior.) You can create a launch template using `aws.ec2.create_launch_template()`, other interfaces to the EC2 API, or the EC2 web console.

[TODO: template parameter overrides are a planned feature. update this when finished.]

If you don't do this, `hostess` just creates a 'scratch' template to launch the instance and deletes it immediately after launch (whether successful or failed).

defaults

The EC2 API requires explicit specification of a lot of parameters to launch an instance. `hostess` wants to make it easy to launch instances, so if you don't explicitly specify some or all of these parameters, it populates them with sensible defaults. You can override the following defaults in `hostess.aws.config.user_config`:

- instance_type (instance type, like 't3a.micro')
- volume_type (root EBS volume type, like 'gp3')
- volume_size (root EBS volume size in GB)

[TODO: complete description of valid `options`]

A couple of default behaviors are worth special consideration:

In [ ]:
```
# First, create a reusable launch template with a few non-default options.
# This creates a launch template in your AWS account, not just a local objec
from hostess.aws.ec2 import create_launch_template

response = create_launch_template(
    template_name='kitty',
    # a cheap burstable instance
    instance_type='t3a.micro',
    volume_size=9,
    # gp3 volumes typically have better sustained performance, and hostess
    # defaults to gp3. However, gp2 volumes tend to have better initial
    # startup and shutdown times, so they're better for a quick demo.
    volume_type='gp2',
    instance_name='kitty',
    tags={'BillCode': 'Customer5', 'Project': 'ADMIN'}
)

# Note that you can't have multiple templates with the same name, so
# this cell will throw a `ClientError` `Exception` if you run it twice.

# If you happen to have run this cell previously and want to create a
# new 'kitty' template with different settings, you'll need to delete
```

In [ ]:
```python
# Now launch an instance using the template. You can specify the 'kitty'
# template in subsequent launches to create identically-configured instances
# in the same security group.

from hostess.aws.ec2 import Instance

# We're passing `connect=True` here to make sure the instance is fully
# booted up before we send the stop instruction in the next section.
# If you send a stop instruction to a booting instance, it can take
# a really long time to stop.
kitty = Instance.launch(template='kitty', connect=True)
```

However, we'll show a preferred workflow here.

2. listing and finding instances

If you have ever used the `awscli` command `aws ec2 describe-instances` to try to find an EC2 instance, you may have noticed that although its output is very complete, it is extremely verbose, deeply nested, and hard to parse. Its filtering options are also somewhat difficult to use. `hostess`'s `ls_instances()` is a much more lightweight alternative that is equally suitable for most use cases.

In [ ]:
```python
# If you simply call `ls_instances()` with no arguments, it will return a tu
# of dicts giving essential information about all your running, pending, sto
# or stopped instances:
# name (if any); public ip (if any); instance id; state (running,
# pending, stopped, terminated); private ip (if any), and keyname (if any).
from hostess.aws.ec2 import ls_instances

instance_info = ls_instances()
instance_info[0:2]
```

In [ ]:
```python
# `ls_instances()` offers a variety of ways to filter your search.
# See the docstring for a full set of options. We'll just describe
# one here: `ls_instances()` understands arbitrary keyword arguments
# as case-insensitive tag filters. For example, if you have a single
# instance named 'kitty', you can find it with:
ls_instances(name='kitty')[0]
```

In [ ]:
```python
# This feature also supports optional regex matching. If you had a
# set of instances named 'pipeline_1', 'pipeline_2', etc., then
# the following command would find them. If you don't (which is
# probably the case), then it will return an empty tuple.
ls_instances(name=r'pipeline_\d', tag_regex=True)
```

3. controlling instance state

Instance has several methods to control an instance's activation state:

- `start()` boots the instance.
- `stop()` shuts the instance down.
- `restart()` shuts the instance down, waits for complete shutdown, and boots it again.
- `terminate()` terminates the instance. **IMPORTANT WARNING:** This permanently and irrevocably deletes an instance and, unless it's specifically configured otherwise, its root volume. `hostess` trusts that you know what you're doing, so `terminate()` doesn't have any special guardrails. `Instance.terminate()` is like `sudo rm -rf`: don't even type it unless you really mean it!

notes:

- These methods don't do anything if the instance is already in (or transitioning to) the requested state.
- `start()`, `stop()`, and `restart()` will raise exceptions on terminated instances.
- `Instance()` also has `wait_until_running()`, `wait_until_stopped()`, and `wait_until_terminated()` methods, which block until the instance reaches the specified state.

In [ ]:
```
# For the sake of demonstration in the next section, shut `kitty``'s
# EC2 instance down and delete `kitty` from the namespace.
# Note that deleting the an `Instance` object like `kitty` does _not_
# terminate the actual EC2 instance associated with it; you need
# to terminate the actual EC2 instance if you don't want it to hang
# around in your account. (We'll discuss that more later.)
kitty.stop()
# Wait until the 'kitty' instance fully shuts down. May take a minute.
kitty.wait_until_stopped()
del kitty
```

4. connecting to an existing instance

`Instance` can connect to an existing instance even more easily than it can launch a new one.

The only required argument to the `Instance` constructor is an identifier for the EC2 instance you'd like to work with. There are three acceptable types of identifier:

- a connectable IP address for the instance, like `"102.31.4.129"`
- the AWS instance identifier, like `"i-0868ad3eeebe16cde"`
- one of the `dicts` returned by `ls_instances()`

Note: If you're connecting to an instance from another EC2 instance, pass `use_private_ip=True` to the `Instance` constructor, and if you're using an IP address as the identifier in this case, make sure it's the private IP.

In [ ]:
```python
# The EC2 instance named 'kitty' exists, but is stopped, and we no longer
# have an `Instance` object for it. We can find the EC2 instance and
# make a new `Instance` for it by using `Instance` along with `ls_instances(
from hostess.aws.ec2 import Instance, ls_instances

kitty = Instance(ls_instances(name='kitty')[0])

# The string representation of an Instance gives its name (if any),
# its instance id, its instance type, its EC2 subnet, and its IP address
# (if it has one, which it generally won't if it's not running).
print(kitty)
print()

# Instance also has a number of 'basic information' attributes, like:
print(
    f"The instance is {kitty.state}, named '{kitty.name}', "
    f"and has the following tags:\n\n{kitty.tags}"
)
```

In [ ]:
```python
# Let's boot `kitty` back up so we can use it.
# It can take a minute or two for an instance, especially a little
# instance like this, to launch its SSH daemon after it boots.
# `connect=True` makes `start()` block until you can establish an SSH
# connection to the instance; here, we're doing this to ensure
# that we'll be able to run commands in the next section.
kitty.start(connect=True)

# Now it's got an IP address again! Good kitty!
kitty
```

5. remote procedure calls

`Instance` supports two main types of remote procedure calls (RPCs):

- shell commands
- Python function calls

5.1. shell commands

With `Instance`, you can run commands as if you were logged into the instance and work with the output of those commands in Python. `Instance` has three primary methods for this. They are all highly configurable, and we don't discuss all their options here. See the documentation for `hostess.subutils.RunCommand` and `hostess.subutils.Viewer` for a full description of options.

- `command()` runs a command in the remote user's default login shell. you can pass a command as a literal string, or construct it from Python arguments (examples below). By default it runs the command asynchronously and returns a `hostess.subutils.Viewer` object you can use to inspect or terminate the process.
- `con()` simulates the experience of typing a command into a console and looking at its output. It blocks until the process exits and pretty-prints any output from the process. the exception.
  - This means that if you run a command that never exits, like `'while sleep 1; do uptime; done'`, `con()` will block until it is interrupted or the connection is severed.
  - `con()` prints "^C" on `KeyboardInterrupt` rather than raising
- `commands()` provides syntactic sugar for constructing list commands.

Note: `hostess` only fully supports `bash`. Some functionality may not work in other shells.

In [ ]:
```python
# Get the full UNIX name information for the instance.
# `hostess` interprets the `a=True` argument as a shell
# switch; this is equivalent to `kitty.command('uname -a')`.
uname = kitty.command('uname', a=True)

# Because `Instance.command()` runs asynchronously by default,
# it's unlikely that you'll get any output in the microseconds
# it takes to get to this next line. You'll probably just see
# that 'uname' is a `Viewer` for a running 'uname -a' process.
uname
```

In [ ]:
```python
# The uname call will probably have completed by the time you
# execute this cell. If not, just run this cell again.
uname
```

In [ ]:
```python
# If you want a more streamlined way to look at the output of a command,
# try `Instance.con()`. This blocks until process exit and iteratively
# pretty-prints its stdout and stderr (with stderr in red).
# It's intended to give the feel of running a command in a console.
# IMPORTANT: By default, in order to keep things nice and light, `con()`
# does not return anything. If you need to use the results of the command
# in subsequent code, pass `_return_viewer=True`.

# To pretty-print all active TCP connections, timestamped, twice:
kitty.con("date; ss -t ; sleep 1; date ; ss -t")
```

In [ ]:
```python
# If you would like to ensure that a command completes before you move on
# to the next part of your code, but don't want it to be loud like `con()`,
# you can call the `.wait()` method of the returned Viewer object or pass
# `_wait=True`.

# get size and usage information for all mounted filesystems
usage = kitty.command("df -h", _wait=True)
print(usage.stdout[0])
```

In [ ]:
```python
# You can access the stdout and stderr of commands you execute
# via the `.out` and `.err` attributes of returned Viewers.

# These are lists of strings. Each element of the lists is an
# individual write to stdout/stderr by the remote process.
# Simple commands that do a thing and exit will generally only have
# one element, because they return their output all at once.
# (This may not be true in cases in which the output is extremely
# large, due to SSH buffering, etc.)
print(f"{len(usage.out)} write(s) to stdout")
```

In [ ]:
```python
# This allows you to use the results of remote shell commands in code.
import re
import pandas as pd

# For example, reformat the contents of `usage.out` into a DataFrame:
rows = [
    re.split(' +', line, maxsplit=5)
    for line in usage.out[0].splitlines()
]
pd.DataFrame(rows[1:], columns=rows[0])
```

In [ ]:
```python
# It also allows you to monitor the results of ongoing processes.
# A silly example:
kitty.command("echo 1 > numbers.txt", _wait=True)
tail = kitty.command("tail -f numbers.txt")
number = 1
while len(tail.out) < 5:
    number += 1
    kitty.command(f"echo {number} >> numbers.txt", _wait=True)
    print(tail.out)
```

In [ ]:
```
# It can sometimes be important to do manual cleanup of backgrounded RPCs.
# The `tail` process that we executed in the previous scell is still
# running, even though we're done with it:
tail.running
```

In [ ]:
```
# To you might want to kill it.
tail.kill()
tail.running
```

In [ ]:
```
# `Instance.commands()` is an easy way to perform a sequence of commands
# without having to enter a monolithic string. A silly example:
kitty.commands(["cd /", "ls"], _wait=True)
```

In [ ]:
```
# This can be used for serious sysadmin stuff.
# A real-world example might look like this:
from itertools import chain

private_repos = ["sensitive_devops", "proprietary_algos", "company_secrets"]
update_commands = chain(
    *[
        (f"ssh-add .ssh/{repo}_deploy", f"cd {repo}", "git pull", "cd ~")
        for repo in private_repos
    ]
)
update_result = kitty.commands(
    ["eval `ssh-agent`", *update_commands], op="and", _wait=True
)

# This won't actually work, of course, because these are just
# hypothetical keys and repos, but you get the idea.
print(f"The directories don't really exist:\n{update_result.err[0]}\n")

# Note that `op="and"` caused `hostess` to chain the long sequence of comman
# with `&&`, which is why `bash` didn't continue after `ssh-add` failed:
print(f"We tried to run\n{update_result.command}")
```

5.2. python function calls

`Instance.call_python()` permits direct invocation of Python functions in any installed interpreter.

interpreter selection and installation

`call_python()` requires a Python interpreter on the remote host.

- By default, it uses the first `python` on the shell $PATH (if any).
- You can explicitly specify the path to an interpreter by passing it as the `interpreter_path` argument.
- If you want to run code in a conda env, you can pass the name of the env as the `env` argument, and `hostess` will find it for you.

The stock Ubuntu AWS image doesn't come with Python. We'd like to demonstrate the special relationship between `call_python()` and Conda, so we're going to install the Miniforge distribution of Python. `Instance` has a convenience method to do *that* for you, too: `install_conda()`.

In [ ]:
```
# This might take a minute or two depending on internet weather.
installation = kitty.install_conda()
installation.wait()
print(installation.out[-1])

# If there were a problem, it might be useful to look at the
# stderr output, which is available in the `.err` attribute of
# `installation`. Note that the miniforge installer writes a
# bunch of stuff to stderr during normal operation, so not
# all of `installation.err` will be useful for debugging.
# >> print(installation.err[-1])
```

making function calls

`call_python()` lets you directly call Python functions on an instance, from simple math functions to top-level handlers for complex applications. A silly example:

In [ ]:
```python
from more_itertools import chunked

# roll 3d6 6 times
rolls = [
    kitty.call_python(
        "random", "randint", (1, 6), splat="*", env="base"
    )
    for die in range(18)
]
# the processes are running asynchronously; wait for them to complete
[r.wait() for r in rolls]

# chunk and sum the results
rolls = chunked([int(r.out[0]) for r in rolls], 3)
stats = [
    kitty.call_python(None, "sum", roll, env="base") for roll in rolls
]
[s.wait() for s in stats]
[int(s.out[0]) for s in stats]
```

a note on performance

It is perfectly appropriate to call computationally inexpensive procedures with `call_python()` for utility, administration, and similar purposes.

However, if you're doing intense computation, `call_python()` is best used to call functions that act as entry points to larger pipelines, or at least perform larger units of work. The preceding example is an exaggeratedly inefficient use of `call_python()`; it sends 24 separate instructions over the network and executes 24 separate Python interpreter processes to add a few random numbers together. For simple operations like this, the time required to make the remote calls and launch the interpreters far outweighs the program run time itself.

Now, a less silly example: quickly turn 'kitty' into a simple web proxy.

In [ ]:
```python
# Write a simple script to retrieve web content.
fetch_script = """
import requests

def fetch_url(url):
    response = requests.get(url)
    response.raise_for_status()
    return response.text
"""
# Write the script to the instance. `literal_str=True` means:
# write `fetch_script` into `fetch.py` as a string instead
# of trying to interpret `fetch_script` as a path to a file.
kitty.put(fetch_script, "fetch.py", literal_str=True)

def fetch(url):
    response = kitty.call_python(
        "/home/ubuntu/fetch.py", "fetch_url", url, env="base", _wait=True
    )
    return ''.join(response.out)
```

In [ ]:
```python
from IPython.display import HTML

# Now you can read Shakespeare in relative privacy.
HTML(fetch("http://shakespeare.mit.edu/midsummer/midsummer.3.1.html"))
```

limitations of `call_python()`

- `call_python()` is versatile but is not designed for passing enormous single arguments. If you want to use `call_python()` in applications that work with large quantities of data, you'll be better off storing the data in files and calling functions that accept their filenames as arguments.
- `call_python()` does not currently implement automated serialization or compression for return arguments. If you want to get serialized objects back from the instance, you will need to ensure that the called function includes a serialization step and then deserialize it yourself locally. This is likely to change in the near future.
    - *Note: if you want to dynamically pass Python objects around in a supervised way, consider looking at the `hostess.station` framework.*

5.3: remote Jupyter Notebook access

Many Python programmers (including us) suffer from a deep attachment to Jupyter. Firing off scripts is lovely, but sometimes you need to roll up your sleeves and dig around in an interactive environment, and while shells are fine, cell persistence, graphical capabilities, and ... well, we probably don't need to talk you into it.

On the other hand, setting up remote Jupyter servers can be a giant hassle, and if you permit HTTP access to them, it's very hard to make them secure. Fortunately, `Instance` includes a streamlined method for launching a Jupyter server and tunneling to it over SSH: `Instance.notebook()` . It's a Notebook there, but here, and just for you.

In [ ]:
```
# kitty doesn't have Jupyter, so we'll need to install it first.
# you should expect this to take 2-3 minutes depending on internet
# weather. Jupyter Notebook has a lot of dependencies. You will
# see output from the `mamba` installation process while it is
# running, though, because of the magic of `con()`.

kitty.con(
    "/home/ubuntu/miniforge3/bin/mamba create -y -n notebook notebook"
)
```

In [ ]:
```python
# There are no mandatory arguments to `Instance.notebook()`,
# although it'll only work that way if you have Jupyter
# installed in a Python environment on your default $PATH.
# we'll show some useful options here for the sake of
# demonstration:

url, tunnel, tunnel_info, launch, stopper = kitty.notebook(
    # what port would you like to use for the local end
    # of the tunnel? it's useful to be able to change this
    # if you want to run multiple Instance-powered notebooks
    # or have a lot of other port forwarding going on.
    # also, if you shut down the Notebook but don't kill your local
    # tunnel (see below) you won't be able to bind anything else
    # to that port.
    # automated tunnel closing is a planned feature, but it
    # won't be perfectly reliable in all cases.
    # defaults to 22222.
    local_port=12358,
    # What port would you like the instance to launch
    # the Jupyter server on? defaults to 8888, the default
    # Jupyter port.
    remote_port=10001,
    # Optional name of a conda environment -- if not specified,
    # just tries to run a jupyter executable from the $PATH.
    env='notebook',
    # What directory would you like Jupyter to use as
    # its root working directory? If not specified, defaults
    # to the remote user's home directory, so the value
    # here is redundant and just for the sake of demonstration.
    # this is useful because it's difficult to 'back out' from
    # the directory Jupyter launches in, so you won't be
    # able to access anything outside your home directory
    # tree by default.
    working_directory='/home/ubuntu',
    # set this to `True` to use JupyterLab instead of Notebook
    lab=False
)

print(f"Click this URL to try kitty's Notebook:\n{url}")
```

tips for `Instance.notebook()`

`notebook()` launches Jupyter daemonized. This means that if you lose your connection for whatever reason, whatever Jupyter might have been doing on the instance will not stop. You can even reestablish a tunnel to the instance and continue working.

It also means that if you kill the launch process (`launch` in the example above), it won't stop Jupyter in the way hitting CTRL-C does when you run it in an interactive session. `notebook()` , however, returns a function that you can call to gracefully shut down Jupyter (`stopper` in the example above). If something's gone very wrong with Jupyter and this doesn't work, take the gloves off and run `Instance.command('pkill jupyter-noteboo')` (not a typo) to kill all of them.

Also, if you *want* Jupyter to shut down when you disconnect, you can pass `kill_on_exit=True to notebook()` , although this isn't guaranteed to work if your local Python process segfaults or something.

`notebook()` also returns a Process object encapsulating the SSH tunnel, along with metadata about the tunnel (`tunnel` and `tunnel_info` in the example above). If something goes wrong with the tunnel, inspecting these can be useful.

Finally, `notebook()` returns a `Viewer` for the process that actually launched Jupyter (`launch` in the example above). Examining this can be useful as a diagnostic for Jupyter-level issues. Unless you pass `kill_on_exit=True` , killing this process won't actually kill Jupyter; it'll just keep you from receiving its console output.

Note: Conventions for controlling Jupyter have changed with Notebook 7. We do not guarantee backwards compatibility with earlier versions of Notebook.

5.4. managing SSH connections

- As you've seen, you normally don't have to manually establish an SSH connection to an instance. It will happen automatically when you try to use `Instance` functionality that requires a connection. However, the `connect()` method can be used to ensure that you *can* connect to the instance, or to 'prep' it so that there's no connection delay on your first command when you get to that part of a program.
    - This method does nothing if a connection is already open.
- You might also want to close an existing connection and open a new one -- for example, if you lost track of a bunch of processes or the connection gets externally disrupted in a weird way. you can do this with the `reconnect()` method.
    - This will immediately terminate all non-daemonized processes executed over the exiting connection and close any established tunnels / forwarded ports.
    - If no connection is currently open, `reconnect()` is equivalent to `connect()`.
- By default, EC2 uses a strict IP allowlist to gate incoming traffic to instances. This provides good security practice that we recommend retaining, and `hostess` mimics this behavior by default. However, this means that if your IP address changes, you won't be able to connect to your instances anymore until you modify their security group settings. This can be a big hassle if you're on a cell network, using a VPN with IP hopping, or simply moving between networks. `Instance.rebase_ssh_ingress_ip()` can quickly solve this issue for you: it modifies the security groups associated with an instance to permit SSH access from your current IP.
    - **IMPORTANT:** This method revokes all other ingress permissions. Do not use it if you need an instance to be accessible to anyone or anything but you. *This behavior may change in the future*.
- If you would like to establish an SSH tunnel on a specific port so that you can access some service on the instance in another application, you can call `tunnel(local_port, remote_port)`. For example, if you launched a Notebook on the instance using the earlier cell, and then dropped connection for some reason, you could reestablish the tunnel with `Instance.tunnel(12358, 10001)` and go back to whatever you were doing in Notebook. Same deal for anything from PostgreSQL (5432) to DOOM (666).

6. clusters

6.1. launching a `Cluster`

On the backend, all on-demand EC2 instance launch requests are 'fleet requests' -- even if they're for a fleet of one. For this reason, we implemented `Instance.launch()` as a thin wrapper around `Cluster.launch()`, so you can just refer to the earlier section on launching instances for a detailed description of most of `Cluster.launch()`'s behavior. The only difference in the signature is that `Cluster.launch` requires a `count` argument specifying the number of instances to launch.

In [ ]:
```
# Launch a Cluster (clowder?) of 4 kitties.
# Note that Cluster appends a number to each of
# their names so that you can tell them apart.
from hostess.aws.ec2 import Cluster

kitties = Cluster.launch(count=4, template="kitty", connect=True)
```

6.2. using an individual `Instance` from a `Cluster`

Each of a `Cluster`'s `instances` is an `Instance` object. There's no difference between these `Instance` objects and any others -- they're not even aware they're part of a `Cluster`. This means that if you like, you can simply work with them individually. In some ways, `Cluster` is just a list of `Instance`s, and you can even use slice notation to access them individually:

In [ ]:
```
# you could also use `kitty["kitty0"]`, or a kitty's `.ip` or `.instance_id`
kitty0 = kitties[0]

print(kitty0)
kitty0.con('uname -a')
```

6.3. running commands on an entire `Cluster`

`Cluster` also lets you do things with all its `Instances` at once. Its `command()`, `commands()`, `con()`, and `call_python()` methods asynchronously call the corresponding method of all its `Instances` with the same arguments and return the results to you in a list.

In [ ]:
```
# `Cluster.command()` returns a list of `Viewers`:
kitties.command('uptime', _wait=True)
```

6.4 mapping commands across a `Cluster`

Sometimes, of course, you don't want to call the same exact command on all your instances -- instead, you'd like to distribute a workload across them. While you can do this by explicitly iterating over `Cluster.instances` , `Cluster` also offers two automated methods for asynchronously mapping arguments across its `Instances` -- `commandmap()` , which calls `Instance.command()` , and `pythonmap()` , which calls `Instance.call_python()` .

These have several legal calling conventions. Several silly examples follow; see the documentation for more details.

Notes

- These examples use `commandmap()` , but `pythonmap()` uses the same conventions.
- By default, `commandmap()` and `pythonmap()` block until all tasks are complete and return a `list of Viewers` . If you pass `wait=False` , they instead return a `ServerPool` object designed for asynchronous interaction with large sets of remote processes. We do not discuss it in detail in this tutorial.

In [ ]:
```python
# You can map sequences of args and kwargs. In this case, the first argument
# is a sequence of sequences of args, the second, `kwargseq`, is an optional
# of kwargs, which, if present, must be the same length as `argseq`.

# example with args:
kitties.commandmap(
    [("echo", i) for i in range(len(kitties))]
)
```

In [ ]:
```python
# Example adding kwargs:
kitties.commandmap(
    ([("head", "/dev/urandom") for _ in range(len(kitties))]),
    ([{'c': (i + 1) * 2} for i in range(len(kitties))])
)
```

In [ ]:
```python
# You can also pass a single string or a single sequence
# of args if you want to pass the same args to all
# instances.
kitties.commandmap(
    "head /dev/urandom",
    [{'c': (i + 1) * 2} for i in range(len(kitties))]
)
```

In [ ]:
```python
# And you can do the same thing with a single mapping for `kwargseq`:
dirs = ("/opt", "/home", "/run", "/usr")
kitties.commandmap([("ls", d) for d in dirs], {'a': True})
```

In [ ]:
```python
# `argseq` and `kwargseq` can be longer than your number of
# of available instances: these methods automatically dispatch
# pending tasks as instances free up. In this example, tasks after
# the first four could be assigned to any instance, so if you run this
# cell multiple times, you may see the instance names change.

# However, you will not see the numbers that the kitties `echo`
# change: the order in which you pass arguments will always be
# preserved in output.
import random
import re

naps = kitties.commandmap(
    [(f"sleep {random.random()}; echo {i}",) for i in range(10)]
)
sleep_regex = r"sleep ([\d.]{4})"
for n in naps:
    print(
        f"{n.out[0]}: {kitties[n.host].name} slept for "
        f"{re.search(sleep_regex, n.command).group(1)} seconds."
    )
```

6.5. creating a `Cluster` from existing instances

Just like `Instance`, you don't have to launch new instances to create a `Cluster`. The easiest way to do this is by using `Cluster.from_descriptions()`, which lets you create a `Cluster` from instance descriptors, including the output of `ls_instances()`.

tips

- You don't have to do this with instances created from a single fleet request -- if you have EC2 instances of different types, you can still use `from_descriptions()` to group them into a `Cluster`. They do, however, need to be in the same AWS Region.
- Alternatively, if you've already created `Instance` objects, you can construct a `Cluster` from them by passing them to the default class constructor, like: `cluster = Cluster(list_of_my_instance_objects)`. Instances *don't* all have to be in the same Region to create a `Cluster` this way.

```
In [ ]: # Delete the `kitties` object from the namespace to ensure we're
        # not using it. Just like `Instance`, deleting a `Cluster` objct
        # doesn't do anything to the actual EC2 instances.
        del kitties

        # Now reassemble the clowder from an `ls_instances` call.
        kitties = Cluster.from_descriptions(ls_instances(name='kitty'))
        kitties

        # Notice that now we have a bonus fifth kitty! It's the instance
        # that we created at the start of this that's still hanging around.
```

6.6. worked example: make 'browse' images of planetary data

`Cluster` can be used to quickly compose distributed processing pipelines, from simple to complex. Here's a very simple example of a pipeline that distributes data download and rendering tasks across multiple EC2 instances.

Although this is a simple example, it's a good example of how to leverage distributed resources; it utilizes I/O pipe size increase from multiple servers by parallelizing downloads, and also minimizes transfer costs by returning only JPEG-compressed data from the instances to you.

6.6.1. prepare the kitties

We'll use `pdr` to read planetary data. Let's first set up a Python environment on all the instances. Like `Instance`, `Cluster` has an `install_conda()` method, which we'll follow up by creating a conda environment.

```
In [ ]: install_commands = []
        # Install miniforge
        install_commands.append(kitties.install_conda(_wait=True))
        # create a Conda environment called 'pdr' that includes the `pdr`
        # and `requests` packages.
        # This might take a few minutes depending on internet weather.
        install_commands.append(
            kitties.commands(
                [
                    "/home/ubuntu/miniforge3/bin/mamba create -n pdr pdr -y",
                    "/home/ubuntu/miniforge3/bin/mamba install -n pdr requests -y"
                ],
                op="and",
                _wait=True
            )
        )
```

In [ ]:
```
# If you encounter any errors, take a look at the stderr output.
# You can do that like this:
# >> [print(f'{i.err[-1]}\n') for i in chain(*install_commands)]

# Note that the Miniforge install script writes to stderr during
# normal operation, so most of its stderr output will not be useful
# for debugging.
```

6.6.2. write a product download and rendering script

This is a very, very simple pipeline -- we just download the data at a URL, open it with `pdr`, and then use `pdr` to dump it out as a JPEG. It's not hard to imagine how you might make this *much* more complex and configurable, though...

In [ ]:
```
script = """
def thumbnail_from_url(url):
    from pathlib import Path

    import pdr
    import requests

    # this loop lets us get the detached metadata files
    for ext in ("IMG", "LBL"):
        response = requests.get(url.replace("IMG", ext))
        response.raise_for_status()
        with open(Path(url).name.replace("IMG", ext), "wb") as stream:
            stream.write(response.content)
    data = pdr.read(Path(url).name)
    data.load("IMAGE")
    data.dump_browse()
    # pdr's standard browse filename pattern
    return f"{Path(url).stem}_IMAGE.jpg"
"""
# Write this script to all of the instances:
kitties.put(script, "fetch_and_render.py", literal_str=True)
```

6.6.3. map source images across kitties

Now, we'll point the kitties at some Mars images hosted by the Planetary Data System's Imaging Node and let them do their thing.

```
In [ ]: folder = "https://pdsimage2.wr.usgs.gov/Missions/Mars_Science_Laboratory/MSL
        files = (
            "0547MH0002590000201525E01_DRLX.IMG",
            "0547MH0002600000201524E01_DRLX.IMG",
            "0547MH0002630000201527E01_DRLX.IMG",
            "0547MH0002640000201526E01_DRLX.IMG",
            "0547MH0003250050201529E01_DRLX.IMG",
        )
        args = ("fetch_and_render", "thumbnail_from_url")
        # distribute URLs across instances
        kwargseq = [
            {'splat': '**', 'env': 'pdr', 'payload': {'url': f"{folder}{file}"}}
            for file in files
        ]
        results = kitties.pythonmap(args, kwargseq)
```

```
In [ ]: # Get the files as bytes...
        # (don't display this list in Jupyter: too much binary gibberish!)
        jpg_blobs = kitties.read([r.out[0] for r in results], "rb")
```

```
In [ ]: # ...and display one of them.
        from io import BytesIO
        from PIL import Image

        # oh look! an oblong!
        Image.open(BytesIO(jpg_blobs[0]))
```

6.7. Controlling `Cluster` state

`Cluster` lets you pass `start`, `stop`, and `terminate` commands to all its `Instances` at once.

We'll terminate our instances at end of the notebook to clean things up, but first let's talk about the extremely useful cost estimation capabilities of `hostess`.

7. cost estimation

There are a variety of costs associated with operating EC2 instances. The `Instance.price_per_hour()` method provides an easy way to estimate many -- although not all -- of these costs. Specifically, it provides estimates for:

- EBS timed storage
- EBS throughput and IOPS
- timed instance usage

The first time you run this method, it'll take a few seconds, because it's retrieving a large price list from AWS. `hostess` caches this list for a week, so subsequent runs will be faster.

`price_per_hour()` returns a `dict` with keys "running" and "stopped". "Running" provides an estimate of how much the instance will cost per hour when it's in the running state; "stopped", an estimate of cost per hour in the stopped state. Prices are in USD.

```
In [ ]: # kitty should cost about a cent per hour while running, and a
        # cent per 8 hours while stopped.
        kitty.price_per_hour()
```

```
In [ ]: # `Cluster` also has a `price_per_hour()` method. These prices
        # should be exactly 5 times the price for kitty alone, because
        # all the kitties are identically configured:
        kitties.price_per_hour()
```

7.1 limitations of `price_per_hour()`

`price_per_hour()` doesn't take the following things into account, which can cause its estimates to be low or high:

- Data out fees.
  - if the instance transfers lots of data out to the open internet, including your local computer, the estimate will be low.
- Savings plans or other cost reductions.
  - If the instance is a spot instance, if you have a relevant capacity reservation, or if you own a reserved instance of the appropriate type, the estimate will be high.
  - If your account and some or all of the resources you are using are eligible for the AWS Free Tier, the estimate will be high.
- CPU credit charges for burstable instances in unlimited mode.
  - If the instance type name begins with a 't' and you are frequently slamming its CPUs, the estimate will be low.
- Timed storage fees for any snapshots or AMIs created from the instance's volumes.

8. cleanup

Now that we're done working with our kitties, we should let them go. If we merely stopped the instances, then they would continue to incur costs of about $0.001 per hour per instance due to fees for their EBS volumes (virtual storage drives), which is tiny but not nothing. To stop them from incurring any costs, we must terminate them.

In [ ]:
```python
import time
# This is unpleasant but necessary.
kitties.terminate()
# Watch them go... (they might not all finish terminating in 40s)
for i in range(10):
    kitties.update()
    print([kitty.state for kitty in kitties])
    time.sleep(4)
```

appendix: breakdown of estimated costs for this Notebook

Running this Notebook as-is should incur approximately 13 cents in AWS costs.

- This estimate assumes that you take extended breaks in the middle of running this Notebook, so that the first `kitty Instance` runs for 4 hours total and the 4 instances launched in the `kitties Cluster` run for 2 hours each.
- This Notebook uses burstable instances, but the processes it executes are not CPU-intensive enough to incur CPU credit charges.
- EBS API call costs are negligible.
- All values in USD.

t3a.micro on-demand instance usage

```
0.0094 / hour * 1 instance * 2 hours = 0.0188
0.0094 / hour * 5 instances * 2 hours = 0.094
```

data out

```
2 MB @ 0.09 / GB = 0.00018
```

gp2 volume usage

```
9 GB @ $0.10 / GB-month * 2 hours / 730 hours/month = 0.0025
45 GB @ $0.10 / GB-month * 2 hours / 730 hours/month = 0.012
```

TOTAL: ~0.127 USD