facebook

Facebook Job Engine

Automation @ scale using Python

Navid Sheikhol – Production Engineer @ FB

Manual vs Automated

- We (almost) never do manual work on servers
- Automation gives us a repeatable way to perform actions
- Testable
- Code reviews to validate our plans

Sample automation

- Millions of jobs and many years of runtime a day

- Workflows that use FBJE:
    - Kernel and firmware upgrades
    - Provisioning of new hardware
    - Auto remediation (FBAR)
    - Distributing SSL certificates
    - Rolling out widely distributed binaries
    - Many.. Many.. Many more

Let me tell you a story

- You have to perform a somewhat complicated workflow across your entire fleet
- For example, you have to upgrade the kernel on thousands of hosts
- Upgrading the kernel takes some time as you have to power cycle and wait for machines to come back up

Let me tell you a story

- You want to be able to monitor the rollout (looking at the logs)
- The whole thing should run unattended, and it could take months
- You want to be notified if there is any problem
- So that you can correct it, then pick up from where it left off

Run a script from a management host

- Maybe you could run a script from your management host
- But this isn't going to scale
- It also means your colleagues won't be able to follow the progress
- And what happens if you need to reboot that management host, or if it hangs in the middle of it

Problems of that approach

- Hardware volatility: machine where automation runs needs to run the entire time
- Visibility: other users may not have visibility over logs and status. Leads to conflicts and duplication of work
- Environment: different depending which person/user runs the automation

Problems of that approach

- Pause/continue: no easy way to pause (i.e. on failure) and resume from same place

- Scalability: single machine will become the bottleneck as infrastructure grows

FBJE

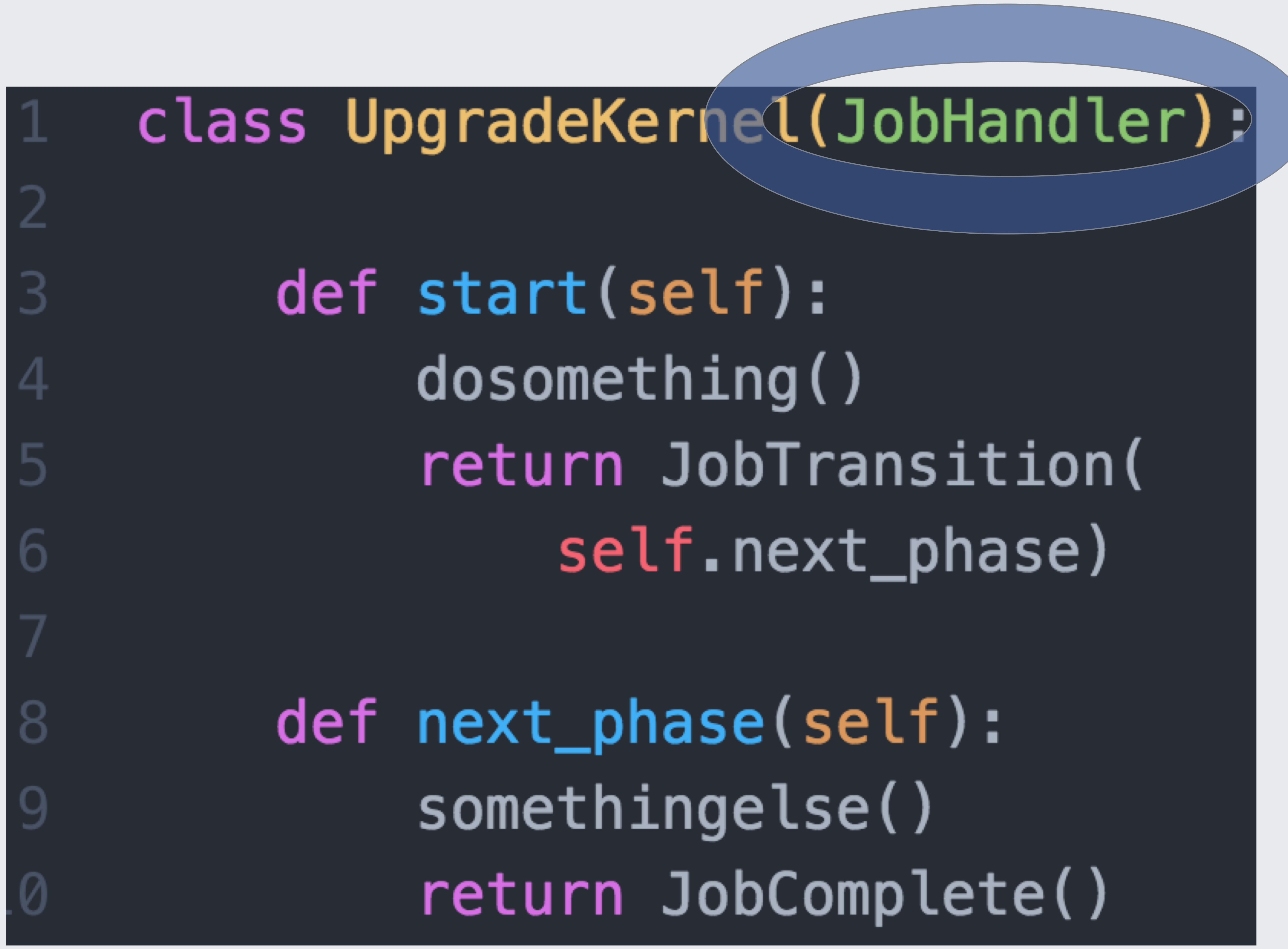
FBJE is a service built at Facebook to implement scalable automation workflows using Python

Job

- A job represents a unit of work, large or small
- Examples: upgrading the kernel on a host, or draining traffic on a cluster
- Jobs can have a parent/child relationship
- **Input** (`entities: Set[str]`)

JobHandler

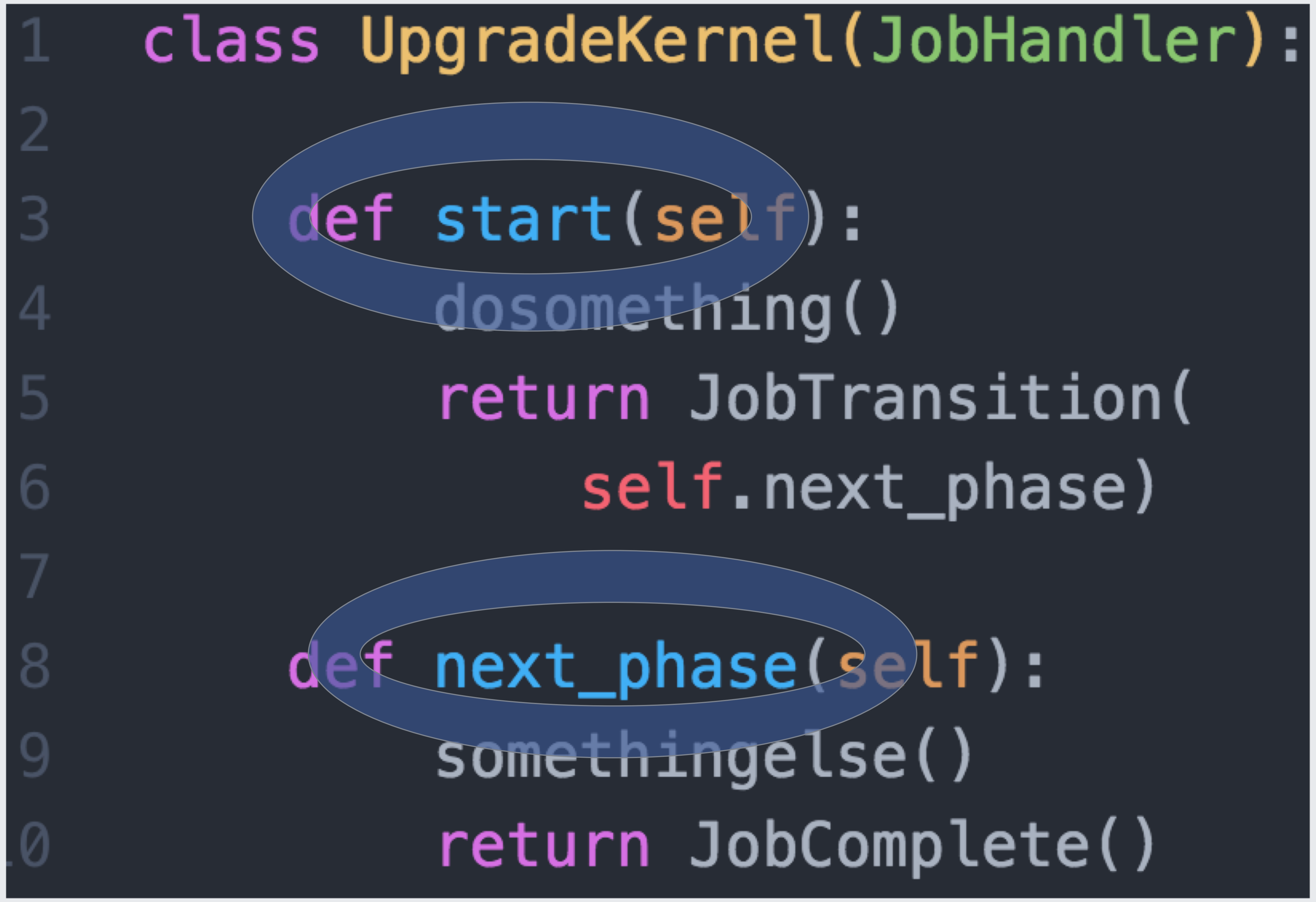
- Python class to extend
- Contains the logic to process a job
- Every class must implement the start() method which is the entry point



```python
class UpgradeKernel(JobHandler):

    def start(self):
        dosomething()
        return JobTransition(
            self.next_phase)

    def next_phase(self):
        somethingelse()
        return JobComplete()
```

Stages

- Each JobHandler method is a stage of your job
- Stages act like save-points
- If there is a failure in a given stage, the job can be retried from that point on (or the beginning)



```
class UpgradeKernel(JobHandler):

    def start(self):
        dosomething()
        return JobTransition(
            self.next_phase)

    def next_phase(self):
        somethingelse()
        return JobComplete()
```

Stage transitions / retries

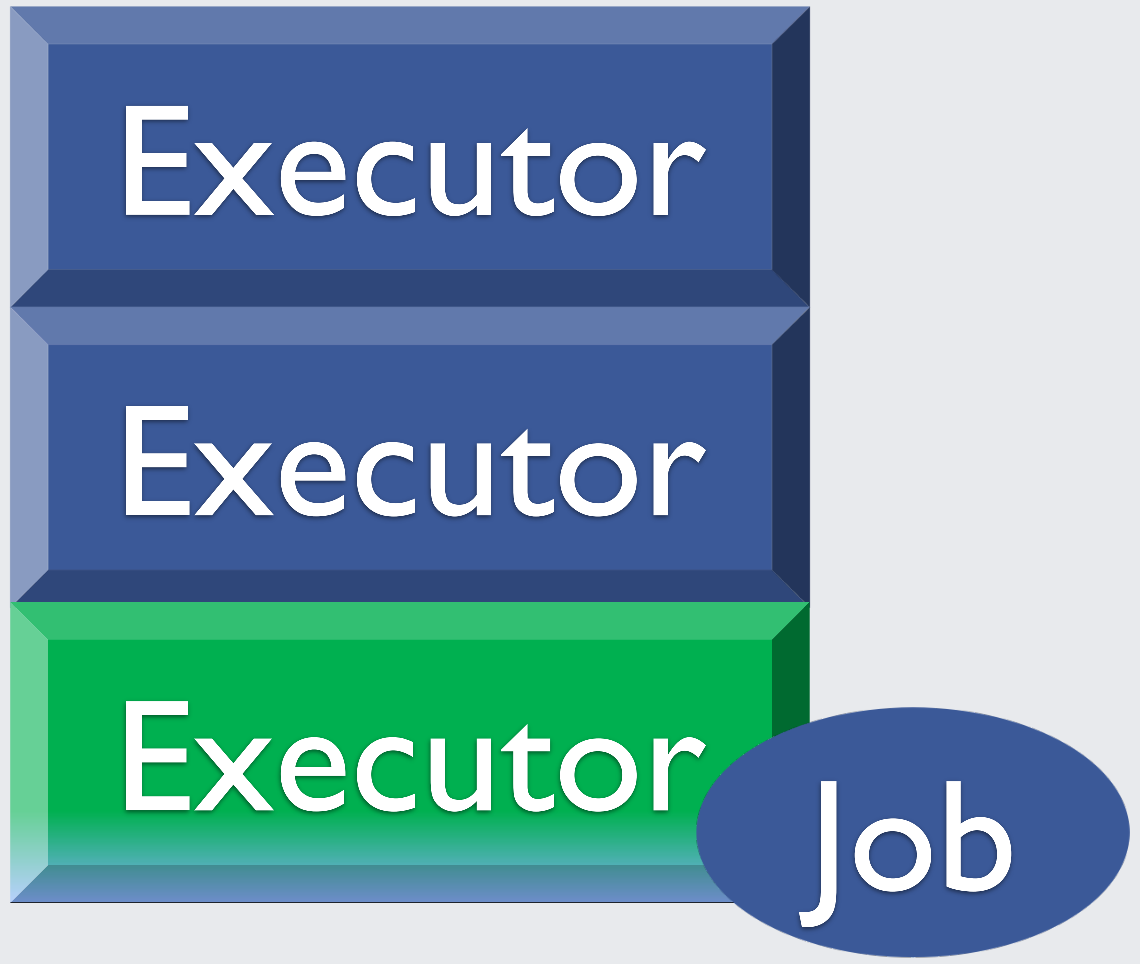
- Every time a job transitions to a new stage we store the information in a DB

- If there is a delay between stages, we will reschedule the job on any available executor when the times comes

- This means that potentially (and quite likely) it will be executed by a different process (no access to prior memory)

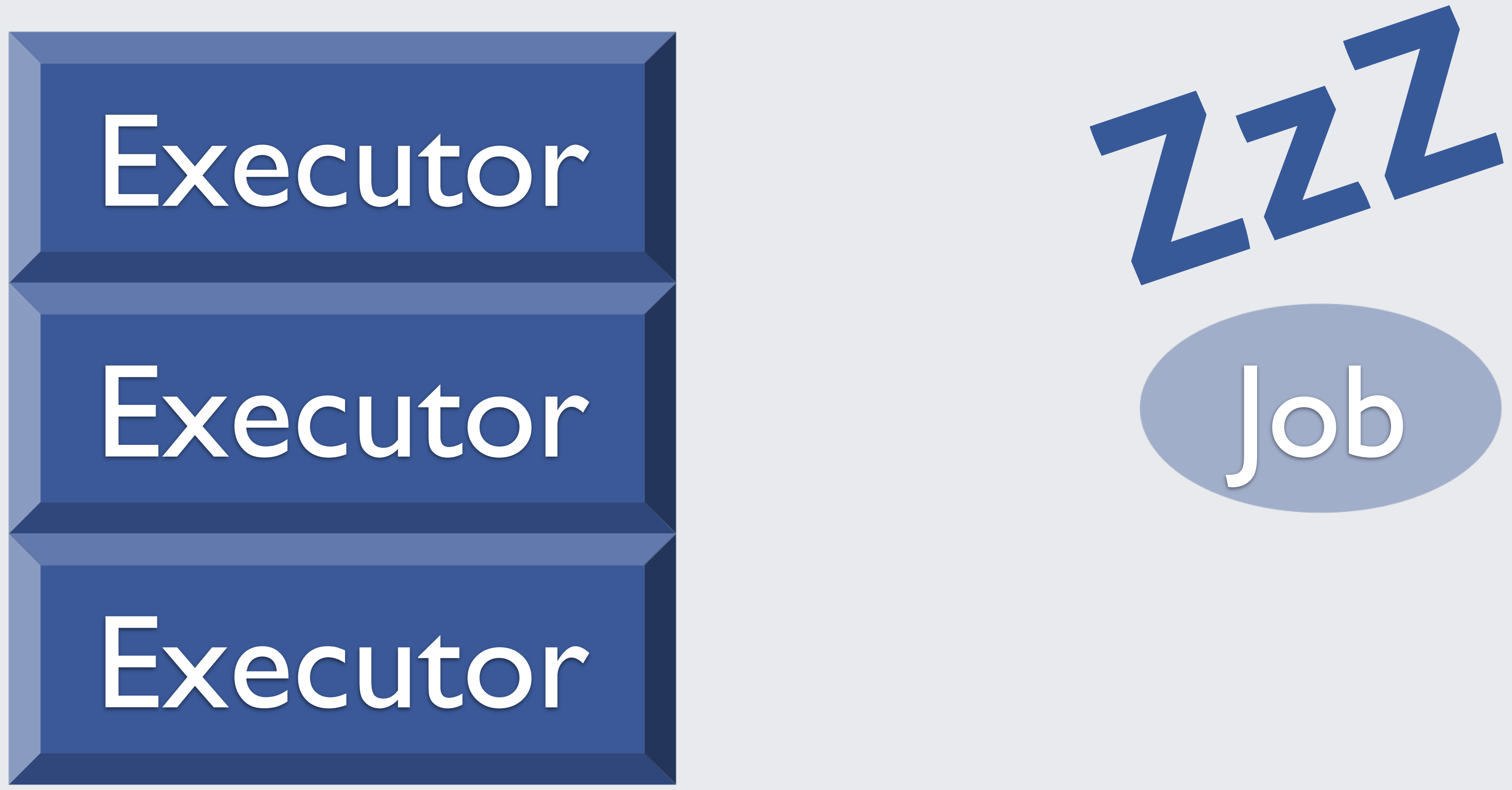
Executors

- Pool of Python processes
- Pick up jobs and execute stages



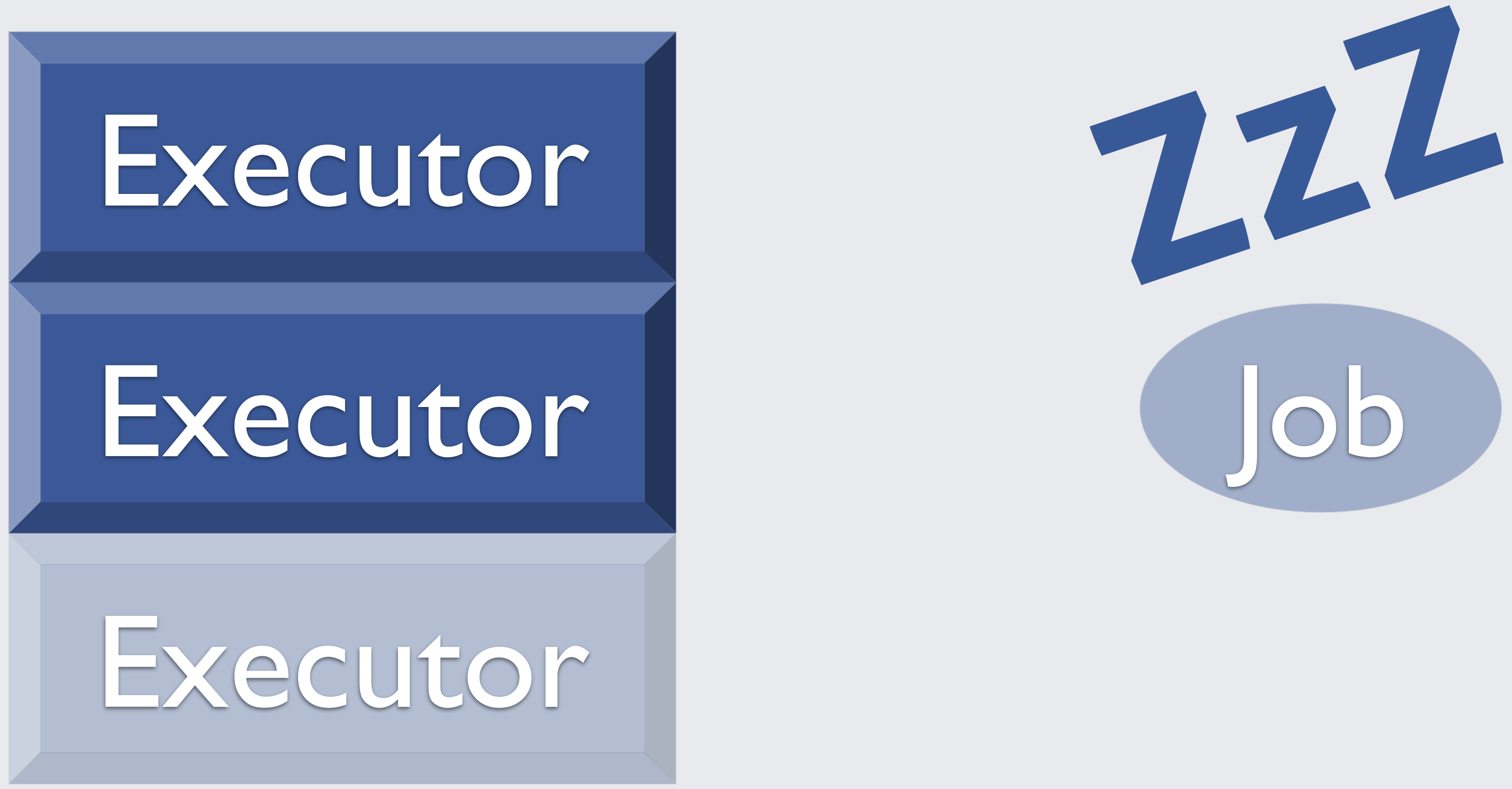
Executors

- Pool of Python processes
- Pick up jobs and execute stages



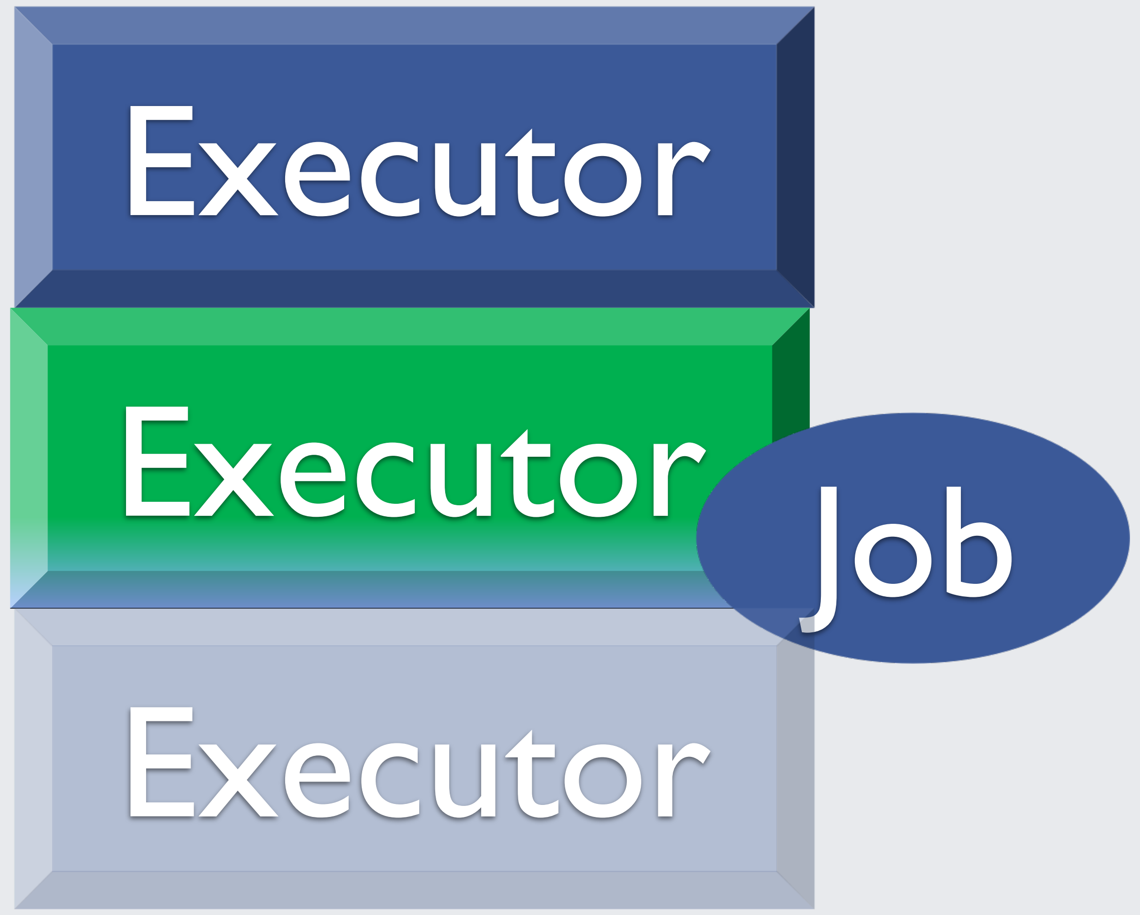
Executors

- Pool of Python processes
- Pick up jobs and execute stages



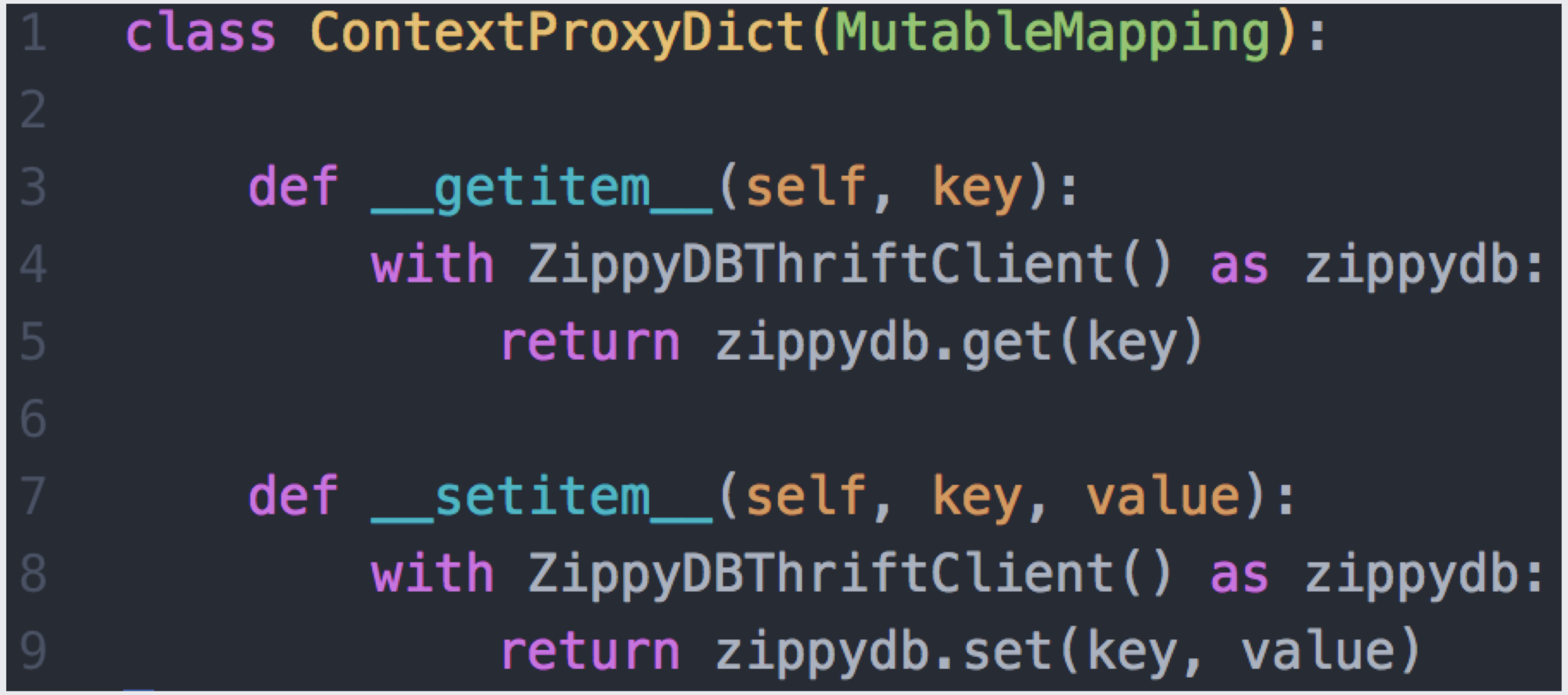
Executors

- Pool of Python processes
- Pick up jobs and execute stages



Context

- The only object that gets persisted across stages is the `self.context` dictionary

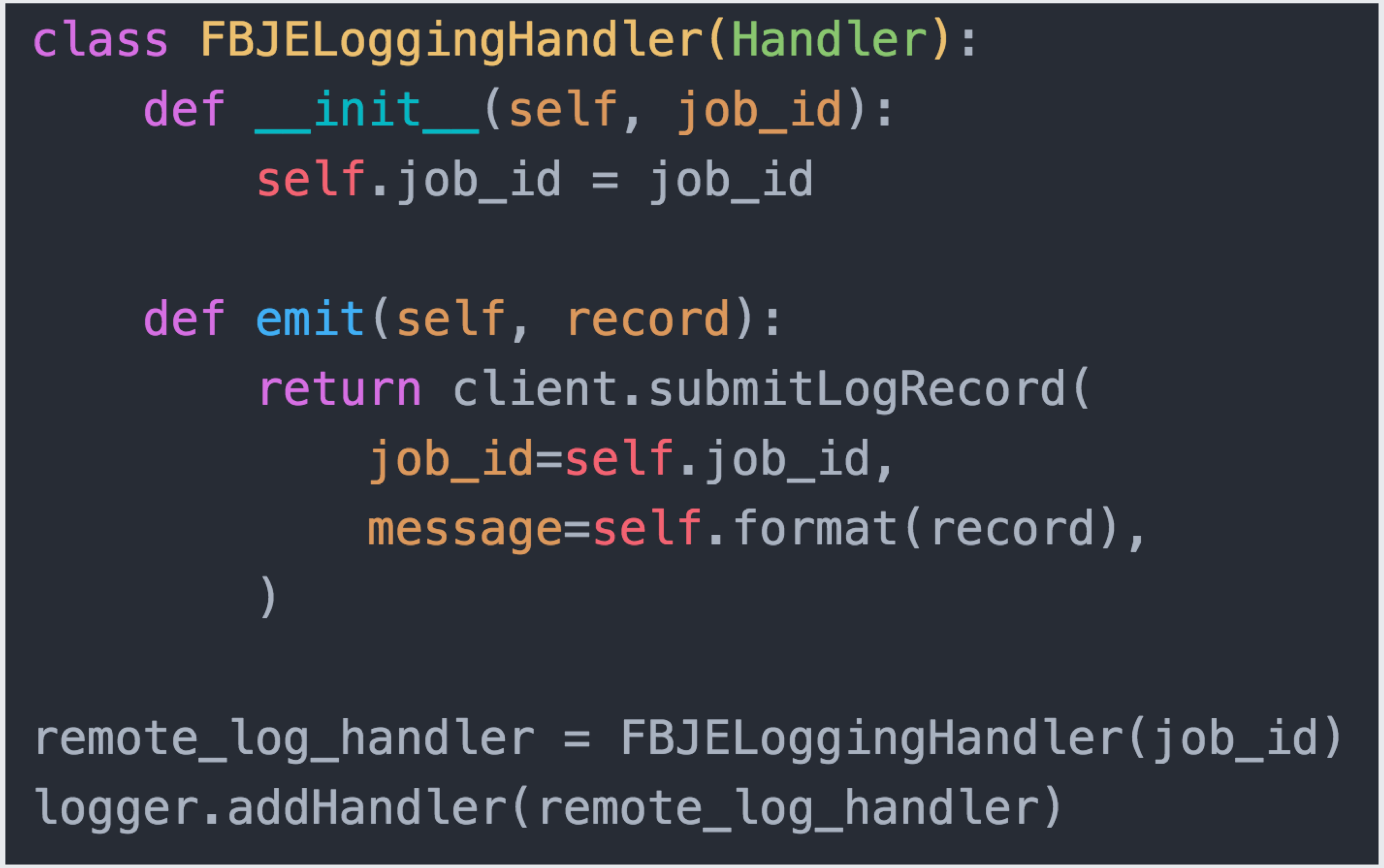


```python
class ContextProxyDict(MutableMapping):

    def __getitem__(self, key):
        with ZippyDBThriftClient() as zippydb:
            return zippydb.get(key)

    def __setitem__(self, key, value):
        with ZippyDBThriftClient() as zippydb:
            return zippydb.set(key, value)
```

- Dictionary-like object: automatically serializes and deserializes objects from a dedicated key/value storage

Logging

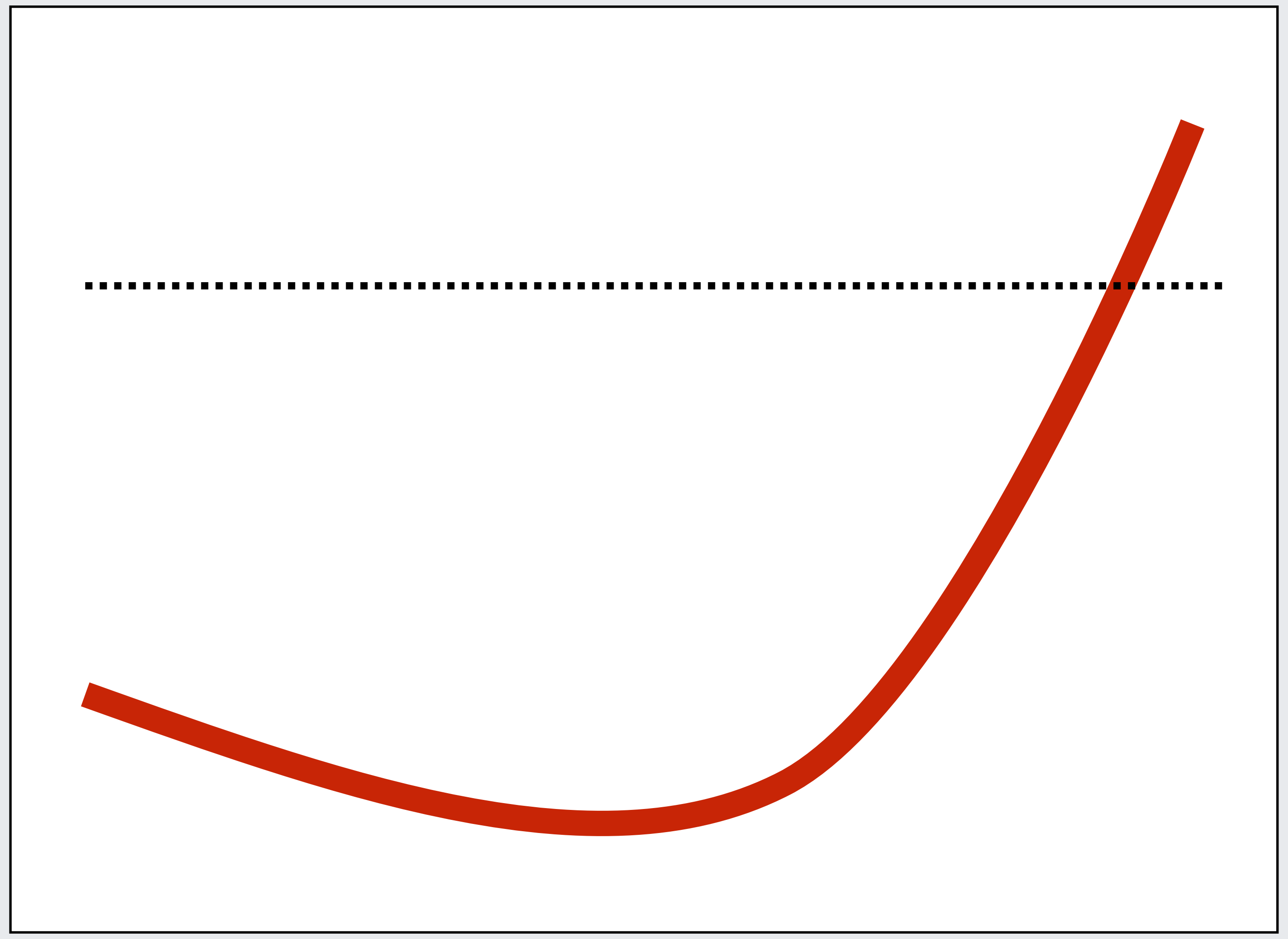
- Logging is an important aspect of FBJE
- Every JobHandler provides a `self.logger` object which forwards logs to a central DB and Hive



```python
class FBJELoggingHandler(Handler):
    def __init__(self, job_id):
        self.job_id = job_id

    def emit(self, record):
        return client.submitLogRecord(
            job_id=self.job_id,
            message=self.format(record),
        )

remote_log_handler = FBJELoggingHandler(job_id)
logger.addHandler(remote_log_handler)
```

Logging

- We auto-generate a number of dashboards and alerts which fire if there is a sudden spike of WARNING/ERROR messages



- Logs are periodically deleted from DB

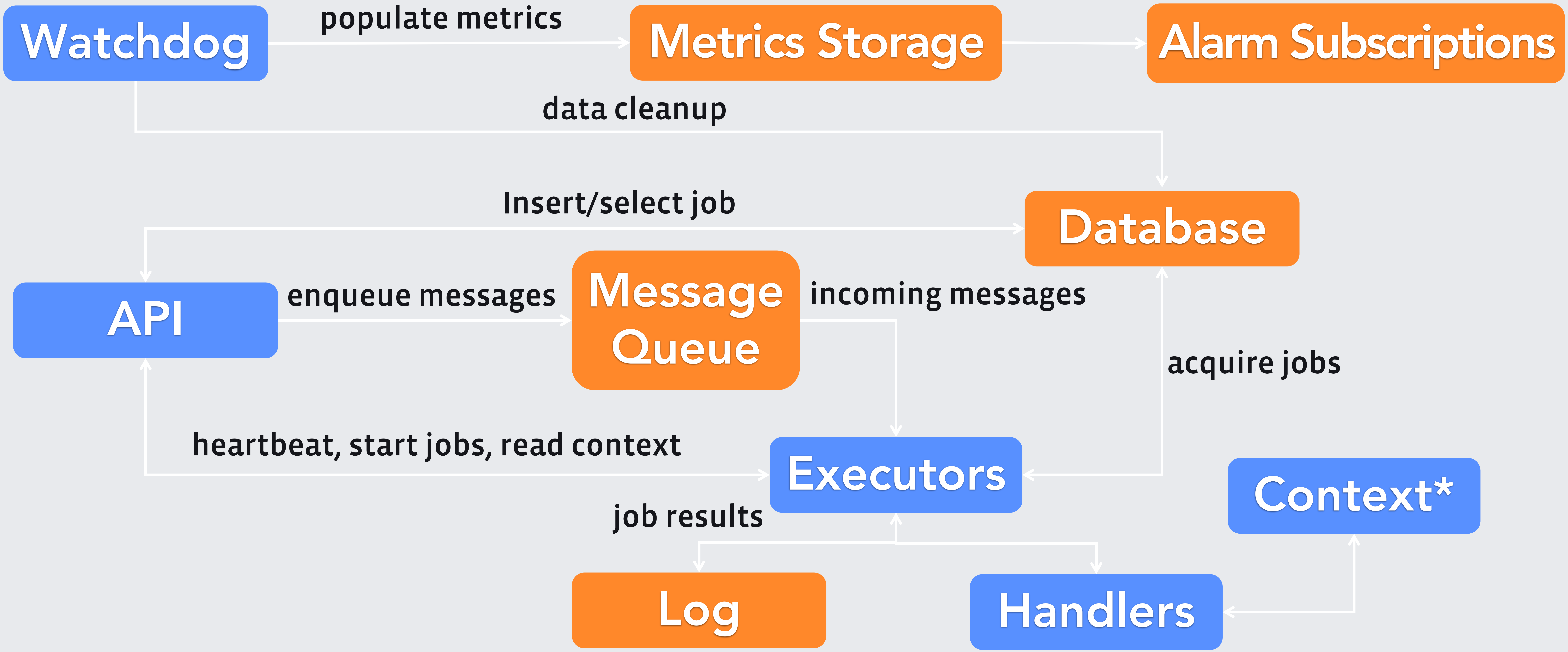
Messaging/Events

- Jobs have the ability to subscribe to event topics and generate events for these topics

- Events are delivered asynchronously to subscribed jobs

- This is important to avoid unnecessary polling which consumes resources

Architecture


Watchdog

populate metrics

Metrics Storage

Alarm Subscriptions

data cleanup

Insert/select job

Database

API

enqueue messages

Message Queue

incoming messages

acquire jobs

heartbeat, start jobs, read context

Executors

Context*

job results

Log

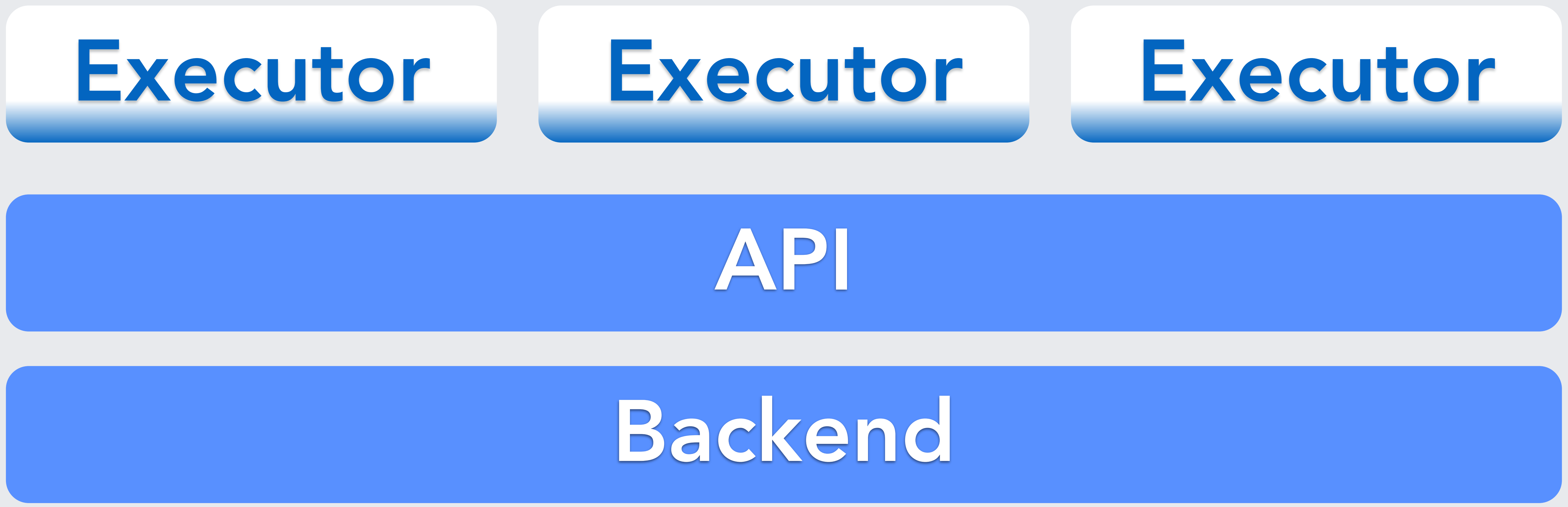
Handlers

Batteries included

- We have integrated FBJE with many services internally so it comes with a lot of freebies
- Dashboards
- Log aggregation (LogView)
- Many default alarms
- Automatic pushes

Lessons learned

- Shared ownership model
    - Executors are “owned” by different teams
    - Base `JobHandler` class owned by FBJE team
    - FBJE backend also owned by FBJE team



Lessons learned

- In the initial design, we used the DB as a queue where executors would pull items to work on

- This became unsustainable as the number of executors grew

- So we migrated to a dedicated message broker which could be compared to RabbitMQ

Lessons learned

- We had many writers synchronously writing to the DB to save the jobs' state

- Contention on database became too high, requiring clients to retry a lot and sometimes fail

- Now we write the updates to a queue/log, and have a fixed number of writers to process updates asynchronously and more optimally

Gracias

facebook

We are hiring!

facebook