ElasticCode

Release 0.0.1

Darren Govoni

Oct 22, 2022

INTRODUCTION

CHAPTER

ONE

OVERVIEW

ElasticCode is a distributed data flow and computation system that runs on transactional messaging infrastructure. It implements the concept of a NVM Networked-Virtual-Machine by distributing logic over networked hardware CPU/GPU processors.

It offers applications and users the following benefits:

- **Persistent Task & Workflow Execution** - Tasks & Workflows persist within the network
- **Reliable Task Execution** - Tasks execution survives failure anomalies, restarts, hardware faults
- **Simplified Workflow Functions** - Parallel, Pipeline, Funnel
- **Powerful Compute Composition** - Build at-scale data and compute flows using CLI, UI or API
- **Streaming Compute** - Real-time streaming compute data flows
- **Secure & Trusted Code Execution** - No client-side code marshalling or serialization. Code is loaded from the network side via git repositories into isolated virtual environments
- **Micro-Scheduling** - Efficient task scheduling and 100% hardware utilization
- **Next-Gen User Interface** - Quickly build out at-scale HPC data flows with simple and intuitive interfaces.

As a platform, ElasticCode is designed so you can build rich, high-performance applications, services and scripts on top. Doing this provides the transparent benefits listed above and makes building powerful compute applications fast and easy.

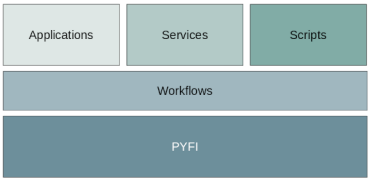


Fig. 1: ElasticCode Platform Architecture

1.1 Managed Compute environment

The entire ElasticCode stack, as a whole, provides a complete "Managed Compute Platform" (MCP) with specialized tooling to support different layers of concern, such as:

- Hardware Compute Resources
- Compute Availability & Scheduling
- Distributed Flows
- Python Processors + Modules & Functions
- Data Logging and Streaming + Real-time & Historical Metrics

ElasticCode is designed as a single, extensible platform for building reliable & persistent computational workflows. It relieves developers from having to know where and when tasks get executed or having to configure client side services. In addition, ElasticCode's multiple API's are designed for users (of all kinds) to build complex, fully-distributed HPC apps and sharable workflows. The platform nature of ElasticCode sets it apart from other libraries and frameworks that only tackle part of the big picture.

1.2 Simple, Parallel Workflows

ElasticCode exposes simple APIs that make writing powerful, distributed workflows fast and easy. A few examples below.

Listing 1: Python API Complex Workflow With pipeline & parallel functions

```
from pyfi.client.api import parallel, pipeline, funnel
from pyfi.client.example.api import do_something_p as do_something

# Create a pipeline that executes tasks sequentially, passing result to next task
_pipeline = pipeline([
   do_something("One"),
   do_something("Two"),
   # Create a parallel structure that executes tasks in parallel and returns the
   # result list
   parallel([
      do_something("Four"),
      do_something("Five"),
   ]),
   do_something("Three")])

# Create another parallel structure using the above pipeline as one of its tasks
_parallel = parallel([
   _pipeline,
   do_something("Six"),
   do_something("Seven")])

# Create a funnel structure that executes all its tasks passing the result to the
# single, final task
_funnel = funnel([
   do_something("Eight"),
```

(continues on next page)

(continued from previous page)

```
    _parallel,
    do_something("Nine")])

# Gather the result from the _funnel and send it to do_something("Four")
print("FUNNEL: ", _funnel(do_something("Four")).get())
```

Listing 2: ElasticCode CLI: Build a distributed, reliable ElasticCode network using simple commands, and then execute a task.

```
# Build out the infrastructure
pyfi add queue -n pyfi.queue1 -t direct
pyfi add processor -n proc1 -g https://github.com/radiantone/pyfi-processors -m pyfi.
↪processors.sample

# Add sockets (not POSIX sockets!) that receive incoming task requests with -c␣
↪concurrency factors (i.e. # of CPUs occupied)
pyfi add socket -n pyfi.processors.sample.do_something -q pyfi.queue1 -pn proc1 -t do_
↪something -c 5
pyfi add socket -n pyfi.processors.sample.do_this -q pyfi.queue1 -pn proc1 -t do_this -c␣
↪8

# Execute a task (can re-run only this after network is built)
pyfi task run --socket pyfi.processors.sample.do_something --data "['some data']"
```

Listing 3: ElasticCode in bash using pipes. Compose pipeline workflows and run parallel tasks using piped output.

```
# Create alias' for the run task commands
alias pyfi.processors.sample.do_something="pyfi task run -s pyfi.processors.sample.do_
↪something"
alias pyfi.processors.sample.do_this="pyfi task run -s pyfi.processors.sample.do_this"

# Pipe some output from stdin to a task
echo "HI THERE!" | pyfi.processors.sample.do_something

# Pipe some text to a task, then append some new text to that output, then send that to␣
↪final task, do_this
echo "HI THERE!" | pyfi.processors.sample.do_something | echo "$(cat -) add some text" |␣
↪pyfi.processors.sample.do_this

# Echo a string as input to two different processors and they run in parallel
echo "HI THERE!" | tee -a >(pyfi.processors.sample.do_something) tee -a >(pyfi.
↪processors.sample.do_this)
```

Listing 4: Easily list out the call graph for any task in your workflow to see where the parallelism occurred

```
$ pyfi ls call --id 033cf3d3-a0fa-492d-af0a-f51cf5f58d49 -g
pyfi.processors.sample.do_something
            └──────────────────────────────────────────────
                                        pyfi.processors.sample.do_something
                                  ␣
```

(continues on next page)

(continued from previous page)

```
↪──────────────────────────────────────────────────────────────────────────────────────
                    pyfi.processors.sample.do_something          pyfi.processors.sample.do_
↪something     pyfi.processors.sample.do_something
```

1.3 Persistent, Reliable Tasks

Task calls in your workflows are completely persisent, meaning they are stored in the ElasticCode network (database) and delivered to the task at the soonest possible time. This depends when the processor hosting the task is available and free to do the compute, but the task call will remain active until it has completed. If the task worker fails for any reason, the task can be retried on another node. These qualities of service are completely invisible to the application or user script.

1.4 High Level Architecture

ElasticCode's high level architecture can be seen in the diagram below. Central to the architecture is the **ElasticCode Model Database** which stores the relational meta-model for the ElasticCode compute network. This database provides the *single source of truth* for the runtime operation of the distributed architecture. Equally as important is the **reliable message broker** which is the heart of ElasticCode's execution model. Workflows execute functions just like any other python code, but those functions trigger persistent requests for ElasticCode to execute a remote task when the compute resources are available. The message broker handles all the runtime orchestration with compute nodes to carry out tasks.

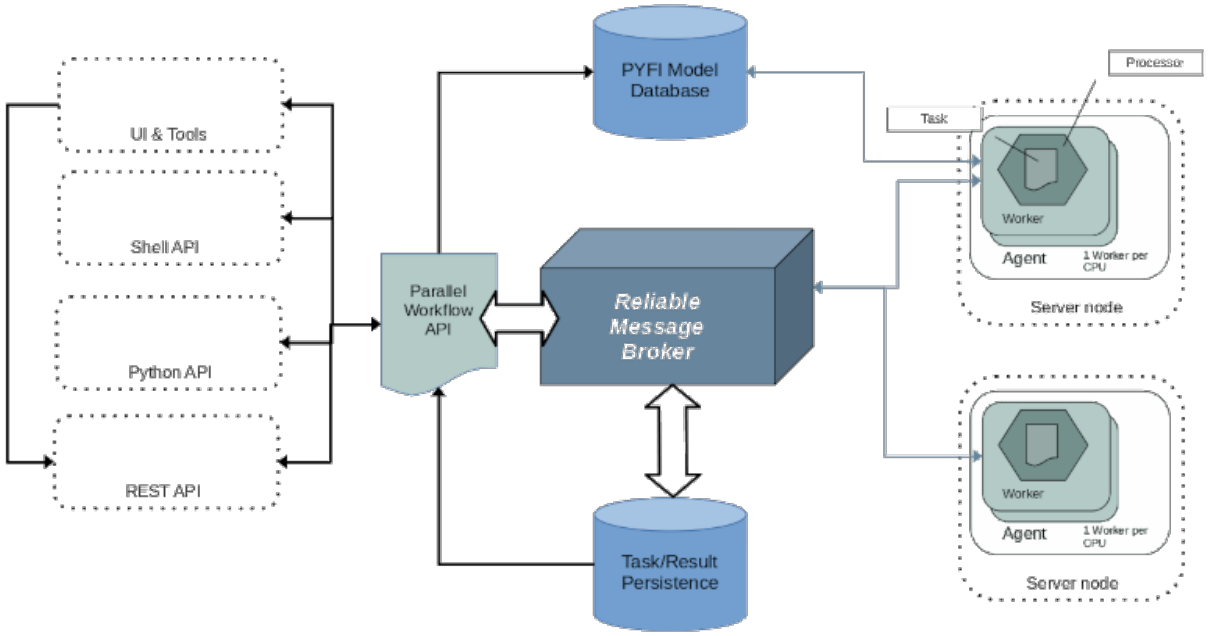


Fig. 2: ElasticCode System Architecture

1.5 Virtual Processors

ElasticCode introduces the notion of **virtual processors** that network together to form a reliable and distributed mesh topology for executing compute tasks.

ElasticCode Processors are object abstractions that capture the location, version and definition of python modules and functions via your own git repository. This trusted code model is important as it establishes the contract between your code, ElasticCode and virtual processors where the code is to be executed. This relationship must be strong and well-defined.

Via the various ElasticCode interfaces (CLI, API, Python etc) you define ElasticCode virtual processors. Agents (a kind of ElasticCode service) running across your network will deploy them and receive tasks to execute their code.

This type of service (or task) mesh architecture allows for fine-grained scalability characteristics that benefit the peformance and stability of the network.

1.6 At Scale Design

ElasticCode is designed to operate "at scale", which means there is a one-to-one correspondence between logic compute units (processors) and physical compute units (CPU cores). This provides a number of obvious and inherent benefits such as hardware redundancy, high-availability, fault-tolerance, fail-over, performance and ease of maintenance.

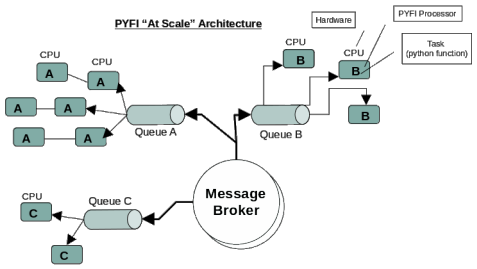


Fig. 3: ElasticCode At-Scale Task/CPU Architecture

1.7 Event Driven

ElasticCode is an event driven architecture from the bottom (data) to the top (ui). This design approach allows it to scale smoothly and not overconsume resources. Messages and notifications are sent when they are available which eliminates the need for *long polling* or similar resource intensive designs. Because ElasticCode is purely event driven, when there are no events, ElasticCode is respectful of system resources and can idle - allowing kernel schedulers and other system tasks to operate as needed.

1.8 Message-Oriented Execution Graphs

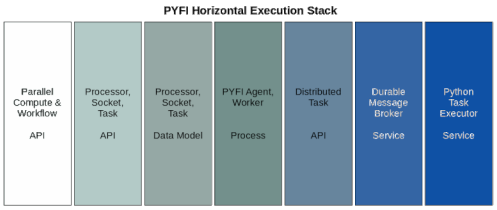
ElasticCode differs from other dataflow engines in that it is fully distributed and runs "at-scale" across heterogeneous infrastructure and computational resources.

It establishes a logical directed-graph (DG) overlay network across compute nodes and executes your custom processor scripts (python, node, bash).

Using the power of reliable, transactional messaging, compute tasks are never lost, discarded or undone. Fault tolerance and load-balancing are intrinsic qualities of ElasticCode and not tacked on as a separate process, which itself would be a failure point.

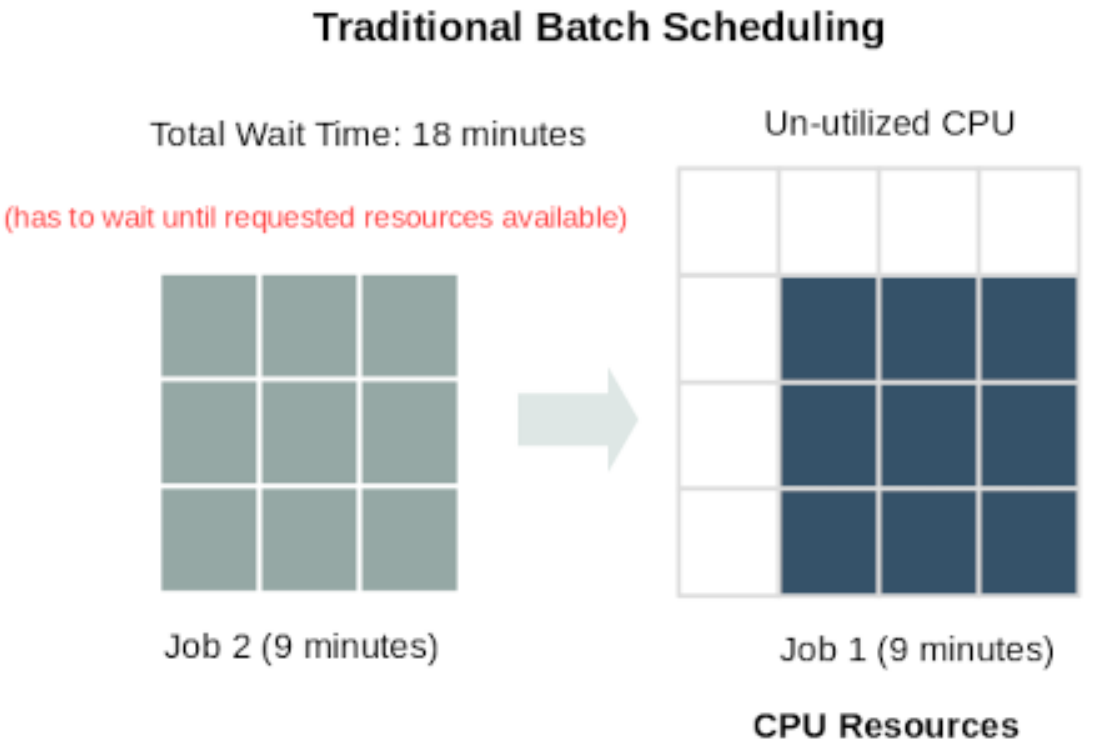
1.9 Execution Stack

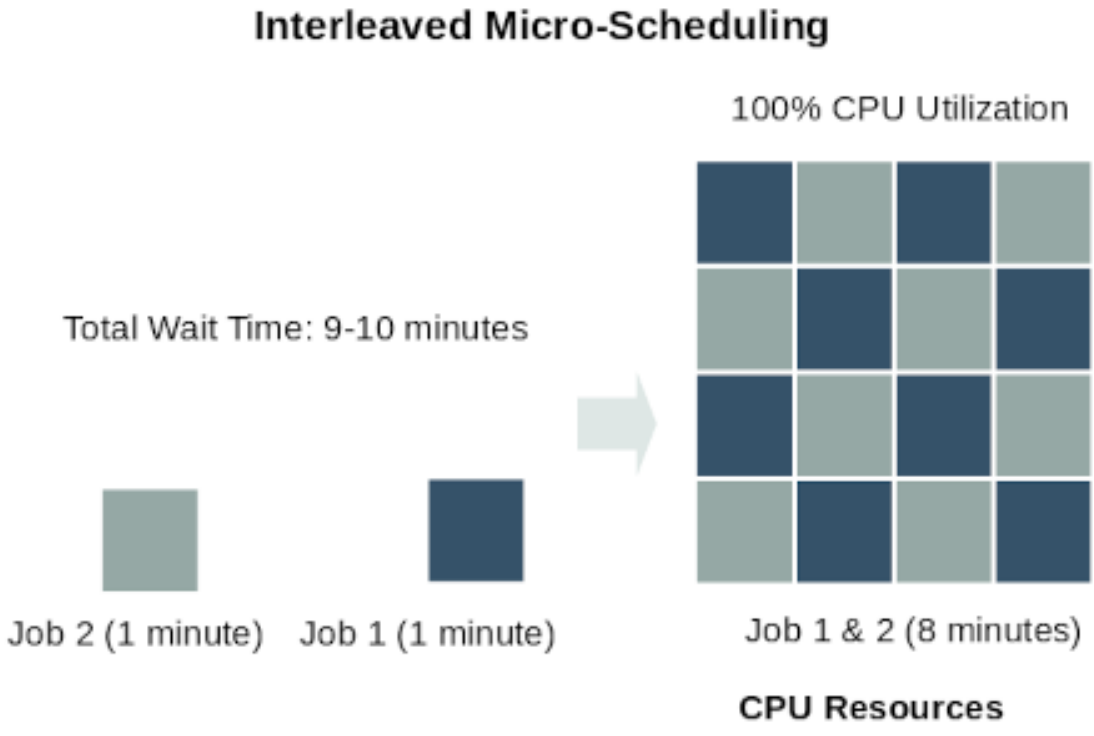
There are various layers within ElasticCode that allow it to scale seamless and expose simple APIs that do powerful things behind the scenes. A quick glance at the lifecycle of a ElasticCode python task is below. Various qualities of service are offered by each layer, most of which are implied during a task invocation.



1.10 Micro-Scheduling

ElasticCode uses a scheduling design that will allow tasks to fully utilize the available CPUs in the ElasticCode network, if processors are created in the ElasticCode database. ElasticCode will never consume more resources than what is requested in its database. Although traditional batch scheduling design allows for blocks of compute resources to be dedicated to one task or flow, it comes at the expense of resource utilization and wait time for other requests. Micro-scheduling seeks to remedy this situation and provide better compute efficiency which means higher task throughput and more satisfied users.


Traditional Batch Scheduling
Total Wait Time: 18 minutes
(has to wait until requested resources available)
Un-utilized CPU
Job 2 (9 minutes)
Job 1 (9 minutes)
CPU Resources


Interleaved Micro-Scheduling
100% CPU Utilization
Total Wait Time: 9-10 minutes
Job 2 (1 minute)
Job 1 (1 minute)
Job 1 & 2 (8 minutes)
CPU Resources

1.11 A True Elastic Compute Platform

ElasticCode provides a set of interacting compute layers that control the location and execution of managed code assets. With ElasticCode, code modules and functions can be loaded at multiple locations and invoked from clients without knowledge of where those functions are or how those functions are executed.

Redundant code (processors) loaded into a ElasticCode network will be able to respond to higher volume of data and requests and thus can scale at will, individually.

Functional tasks (processors hosting code) are fronted by durable queues that deliver reliable invocations when those functions are present on the network, regardless of their exact location. This allows the system to be resilient to hardware or network changes, as well as influence by schedulers that might change the location of functions (processors) to re-balance the resources across the network.

All of this underlying management, hardware arriving and departing, services starting and stopping, processors moving from one host to another (or failing), is completely invisibile to the applications and clients using the system. To them, function calls will always, eventually be executed, if not immediately, in the near future when compute resources allow it.

1.12 System Benefits

The ElasticCode platform provides numerous benefits, only some of which are below.

- **A single, purpose-built platform** that addresses end-to-end managed compute from the CPU to the end user. Compared to cobbled together frameworks.
- **Data flow and data streaming support**
- **Real-time observable data** across your compute resources
- **DevOps out-of-the-box** - ElasticCode integrates directly with GIT allowing your existing code management practices to be used.
- **Elastic, At-Scale** - ElasticCode is an elastic infrastructure, meaning that it scales up and down on-the-fly. Code can be moved across hardware locations at any time without data loss.
- **Extensible** - ElasticCode is designed to be extended and specialized to your needs. Both the UI and the core platform is open and leverages modern framework design patterns to easily build on top of.

1.13 Ecosystem of Supported Roles

The ElasticCode compute environment is a seamless collaboration across disciplines with powerful, out-of-the-box tooling for everyone to manage their concerns, independent of the whole. Let's quantify the previous sentence some. Let's say you are in the middle of running a lengthy workflow, but elsewhere in the grid, hardware administrators need to replace hardware some of your tasks might be running on. With ElasticCode, your workflow would simply pause if it cannot find an active ElasticCode processor hosting the task (python function) it needs and when the hardware admins bring new hardware online, the ElasticCode agents resume and your workflow would continue running where it left off, seamlessly. Sounds amazing but it's true!

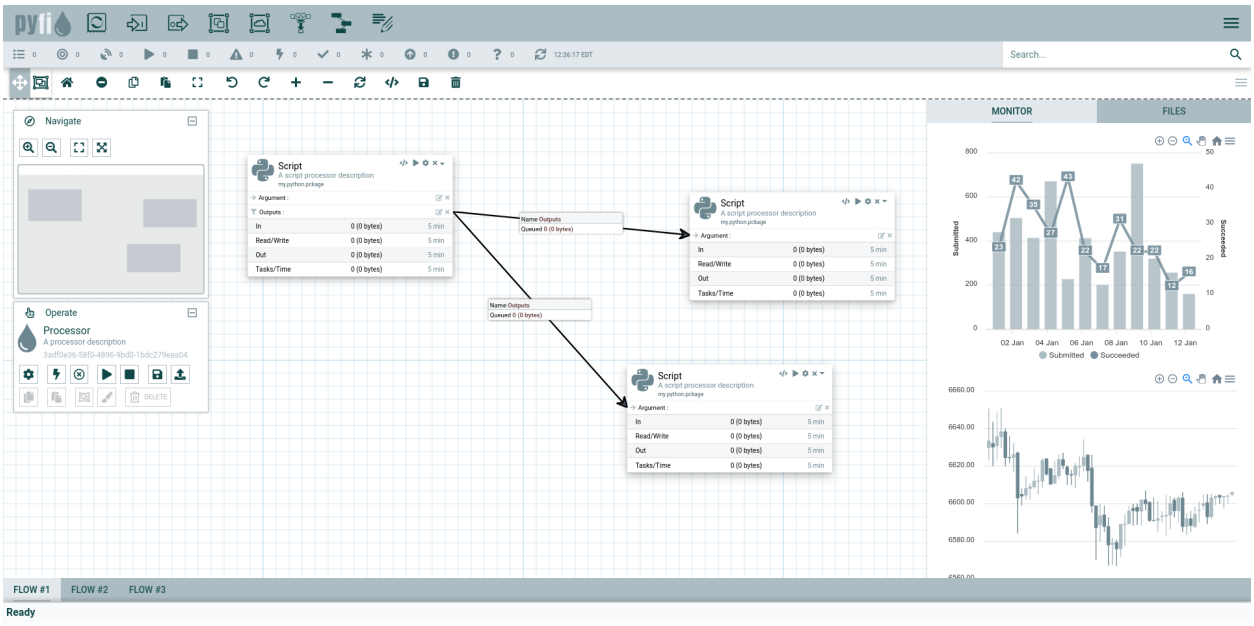
Some of the roles that might participate in a ElasticCode network, directly or indirectly.

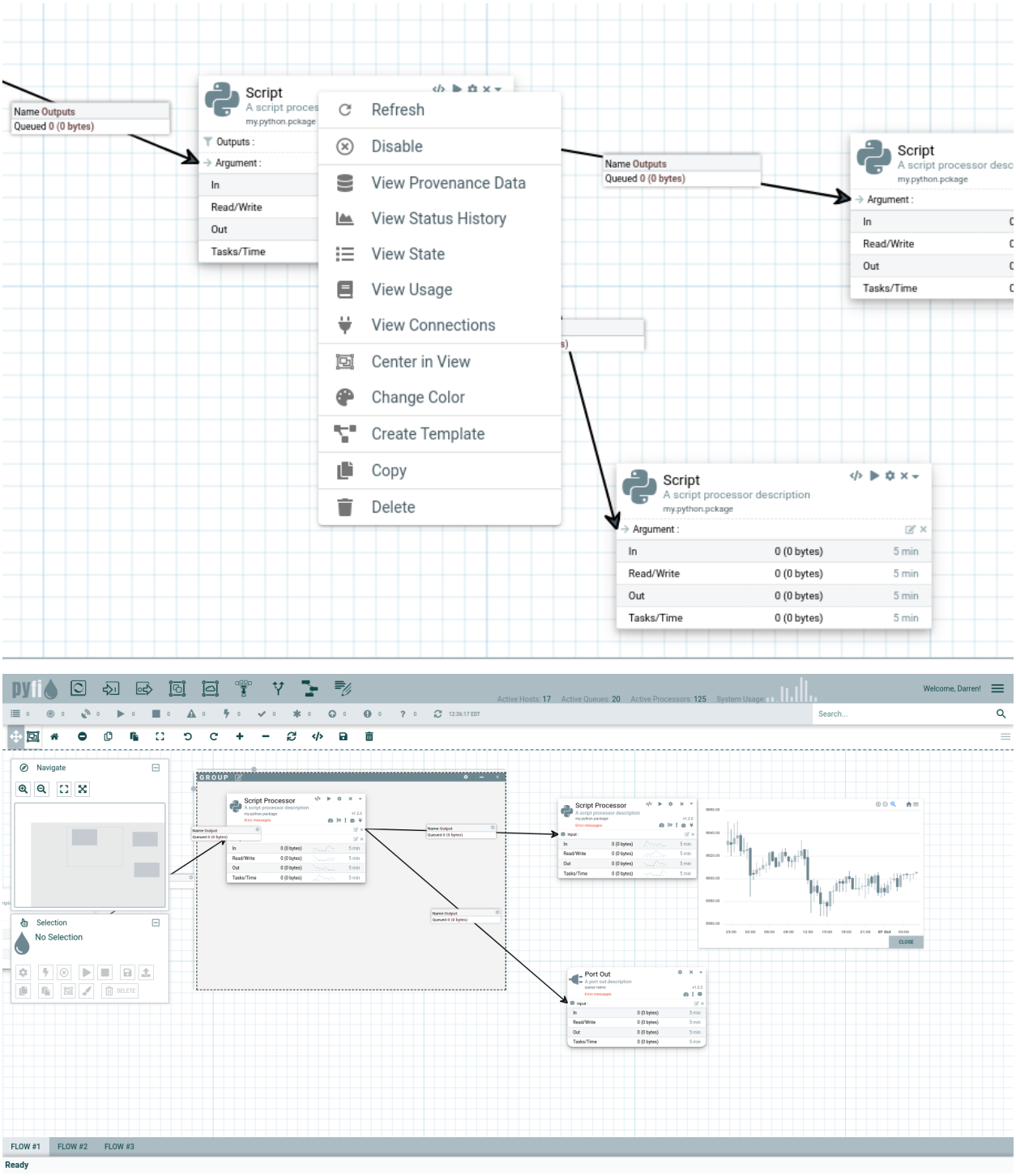
- Hardware Admins
- Infrastructure Admins
- Compute Admins

- Data Admins
- Code Repository Owners
- End Users

1.14 Powerful, Next-Gen UI

ElasticCode's user interface is a powerful, next-gen no-code application that enpowers anyone to create fast, parallel workflows across ElasticCode's distributed task mesh.




Name Outputs
Queued 0 (0 bytes)
Script
A script proces
my.python.pckage
Outputs :
Argument :
In
Read/Write
Out
Tasks/Time
Refresh
Disable
View Provenance Data
View Status History
View State
View Usage
View Connections
Center in View
Change Color
Create Template
Copy
Delete
Name Outputs
Queued 0 (0 bytes)
Script
A script processor desc
my.python.pckage
Argument :
In
Read/Write
Out
Tasks/Time
Script
A script processor description
my.python.pckage
Argument :
In 0 (0 bytes) 5 min
Read/Write 0 (0 bytes) 5 min
Out 0 (0 bytes) 5 min
Tasks/Time 0 (0 bytes) 5 min
Navigate
Selection
No Selection
GROUP
Script Processor
Port Out
FLOW #1
FLOW #2
FLOW #3
Ready

CHAPTER

TWO

GOALS

As the name suggests, ElasticCode is a spiritual offshoot of Apache NIFI except built using a python stack for running python (and other scripting languages) processors. However, ElasticCode is designed to be more broad in terms of design and scope which we will discuss below.

Some important design goals for this technology are:

1. **Fault-Tolerant** - ElasticCode runs as a distributed network of logical compute processors that have redundancy and load-balancing built in.
2. **At-Scale** - This phrase is important. It indicates that the logical constructs (e.g. pyfi processors) run at the scale of the hardware (e.g. CPU processors), meaning there is a 1-1 correlation (physical mapping) between hardware processors and pyfi processors.
3. **Secure** - All the functional components in ElasticCode (database, broker, storage, cache) have security built in.
4. **Dynamic** - The topology and behavior of a ElasticCode network can be adjusted and administered in real-time without taking down the entire network. Because ElasticCode is not a single VM controlling everything, you can add/remove update components without negatively impacting the functionality of the system.
5. **Distributed** - As was mentioned above, everything in ElasticCode is inherently distributed, down to the processors. There is no physical centralization of any kind.
6. **Performance** - ElasticCode is built on mature technology stack that is capable of high-throughput message traffic.
7. **Reliability** - The distributed queue paradigm used by ElasticCode allows for every processor in your dataflow to consume and acknowledge message traffic from its inbound queues and write to outbound queues. These durable queues persist while processors consume messages off them.
8. **Scalability** - Processors can scale across CPUs, Machines and networks, consuming message traffic off the same or multiple persistent queues. In fact, ElasticCode can auto-scale processors to accommodate the swell of tasks arriving on a queue. In addition, flow processors will be automatically balanced across physical locations to evenly distribute computational load and reduce local resource contention.
9. **Pluggable Backends** - ElasticCode supports various implementations of backend components such as message (e.g. RabbitMQ, SQS) or result storage (SQL, Redis, S3) in addition to allowing you to implement an entire backend (behind the SQL database) yourself.
10. **Real-time Metrics** - ElasticCode processors will support real-time broadcasting of data throughput metrics via subscription web-sockets. This will allow for all kinds of custom integrations and front-end visualizations to see what the network is doing.
11. **Data Analysis** - One of the big goals for ElasticCode is to save important data metrics about the flows and usages so it can be mined by predictive AI models later. This will give your organization key insights into the movement patterns of data.

12. **GIT Integration** - All the code used by processors can be pulled from your own git repositories giving you instant integration into existing devops and CM processes. ElasticCode will let you select which repo and commit version you want a processor to execute code from in your flows.

CHAPTER

THREE

USE CASES

There are a wide variety of use-cases ElasticCode can address, a few of which are listed below.

1. **Enterprise Workflow Automation** - ElasticCode can design and execute dynamic workflows across heterogenous enterprise, leveraging a variety of data sources and services.
2. **High Performance Computing** - ElasticCode's support for real-time streaming compute and parallel workflow execution lends itself to big-data and compute intensive tasks.
3. **Enterprise DevOps** - DevOps involves automated and repeatable pipelines for building software assets. ElasticCode's flow models and distributed compute is a perfect fit for custom DevOps.
4. **IoT and Factory Automation** - Orchestrating across connected devices or machinery in a factory is easy to model with ElasticCode due to it's dynamic and ad hoc workflow capability. Custom scripting allows for easy integration into existing device APIs.
5. **AI & Machine Learning Modelling** - Generating effective AI models requires obtaining and cleaning data from various sources, feature extraction, merging and training epochs. This is naturally a multi-step process that can be done visually with ElasticCode's visual modelling tools.
6. **Simulation** - Simulation seeks to model real world processes and given a set of inputs, determine or predict certain target variables. These models are typically designed as a network of connected dependencies or entities along with environmental conditions that affect the simulation.
7. **Decision Systems & Analytics** - State-transition modelling is technique used at major companies that have to make important stochastic financial decisions using key business metrics. ElasticCode's visual modeling and streaming compute capability allow for such models to be easily designed and customized, fully integrating into company databases, spreadsheets, accounting systems or other data sources.

CHAPTER

FOUR

INSTALL

CHAPTER

FIVE

QUICKSTART

5.1 Bringing up the Stack

```
$ docker-compose up
```

5.2 Configuring Lambda Flow

```
$ flow --config
Database connection URI [postgresql://postgres:pyfi101@phoenix:5432/pyfi]:
Result backend URI [redis://localhost]:
Message broker URI [pyamqp://localhost]:
Configuration file created at /home/user/pyfi.ini
```

5.3 Initialize the Database

```
$ flow db init
Enabling security on table action
Enabling security on table event
Enabling security on table flow
Enabling security on table jobs
Enabling security on table log
Enabling security on table privilege
Enabling security on table queue
Enabling security on table queuelog
Enabling security on table role
Enabling security on table scheduler
Enabling security on table settings
Enabling security on table task
Enabling security on table user
Enabling security on table node
Enabling security on table processor
Enabling security on table role_privileges
Enabling security on table user_privileges
Enabling security on table user_roles
Enabling security on table agent
```

(continues on next page)

(continued from previous page)

```
Enabling security on table plug
Enabling security on table socket
Enabling security on table call
Enabling security on table plugs_queues
Enabling security on table plugs_source_sockets
Enabling security on table plugs_target_sockets
Enabling security on table sockets_queues
Enabling security on table worker
Enabling security on table calls_events
Database create all schemas done.
```

5.4 The Flow CLI

```
$ flow
Usage: flow [OPTIONS] COMMAND [ARGS]...

Flow CLI for managing the pyfi network

Options:
--debug         Debug switch
-d, --db TEXT   Database URI
--backend TEXT  Task queue backend
--broker TEXT   Message broker URI
-i, --ini TEXT  Flow .ini configuration file
-c, --config    Configure pyfi
--help          Show this message and exit.

Commands:
add        Add an object to the database
agent      Run pyfi agent
api        API server admin
db         Database operations
delete     Delete an object from the database
listen     Listen to a processor output
ls         List database objects and their relations
node       Node management operations
proc       Run or manage processors
scheduler  Scheduler management commands
task       Pyfi task management
update     Update a database object
web        Web server admin
whoami     Database login user
worker     Run pyfi worker
```

5.5 Creating Your First Flow

Let's look at the sequence of CLI commands needed to build out our flow infrastructure and execute a task. From scratch! First thing we do below is create a queue. This provides the persistent message broker the definition it needs to allocate a `message queue` by the same name for holding task messages.

Next we create a processor, which refers to our gitrepo and defines the module within that codebase we want to expose. It also defines the host where the processor should be run, but that is optional. We specific a concurrency value of 5 that indicates *the scale* for our processor. This means it will seek to occupy 5 CPUs, allowing it to run in parallel and respond to high-volume message traffic better.

Then we create sockets and attach them to our processor. The socket tells pyfi what specific python function we want to receive messages for and what queue it should use. Lastly, it indicates what processor to be attached to.

Finally, we can run our task and get the result.

```
$ flow add queue -n pyfi.queue1 -t direct
$ flow add processor -n proc1 -g https://github.com/radiantone/pyfi-processors -m pyfi.
↪processors.sample -h localhost -c 5
$ flow add socket -n pyfi.processors.sample.do_something -q pyfi.queue1 -pn proc1 -t do_
↪something
$ flow add socket -n pyfi.processors.sample.do_this -q pyfi.queue1 -pn proc1 -t do_this
$ flow task run --socket pyfi.processors.sample.do_this --data "['some data']"
Do this: ['some data']
```

5.5.1 Creating Sockets

Sockets represent addressable endpoints for python functions hosted by a processor. Remember, the processor points to a gitrepo and defines a python module within that repo. The socket defines the task (or python function) within the processor python module. Thus, a single processor can have many sockets associated with it. Sockets also declare a queue they will use to pull their requests from. This allows calls to tasks to be durable and reliable.

The following extract from the above flow defines a socket, gives it a name `pyfi.processors.sample.do_something`, declares the queue `pyfi.queue1`, associates it with processor named `proc1` and represents the python function/task `do_something`.

```
$ flow add socket -n pyfi.processors.sample.do_something -q pyfi.queue1 -pn proc1 -t do_
↪something
```

5.5.2 Defining Socket Functions

Once you've built out your flow and infrastructure to support it, you can create convenient types that represent your python functions via the Socket class.

For the parallel flow above, we import the .p (or partial) signature from this file, which comes from our Socket we created earlier named `pyfi.processors.sample.do_something`. Remember, the socket captures the module (from its parent Processor) and function name within that module you want to run. Think of it like an endpoint with a queue in front of it.

We take one step further in the file below and rename Socket class to Function simply as a linguistic preference in this context.

```
from pyfi.client.api import Socket as Function

do_something = Function(name='pyfi.processors.sample.do_something')
do_something_p = do_something.p

do_this = Function(name='pyfi.processors.sample.do_this')
do_this_p = do_this.p
```

Once we've created our function definitions above, we can use them like normal python functions as in the parallel workflow below!

5.5.3 Executing Socket Functions

Executing socket functions from python is very easy. Since we can create the socket ahead of time, we only need to refer to it by name as above.

```
from pyfi.client.examples.api import do_something_p as do_something

do_something("Some text!")
```

The just invoke the function reference as you normally would. If you are using the function within a parallel API structure such as `parallel`, `pipeline`, `funnel` etc then you should use the `partial` (.p, _p) version of the function signature. This allows ElasticCode to add arguments to the task when it is invoked. The invocation is deferred so it doesn't happen at the time you declare your workflow. The reason is because your task will execute on thos remote CPU at a time when the workflow reaches that task. So the .p partial is a `signature` for your task in that respect.

5.6 Running a Parallel Workflow

```
from pyfi.client.api import parallel, pipeline, funnel
from pyfi.client.example.api import do_something_p as do_something

# Create a pipeline that executes tasks sequentially, passing result to next task
_pipeline = pipeline([
   do_something("One"),
   do_something("Two"),
   # Create a parallel structure that executes tasks in parallel and returns the
   # result list
   parallel([
      do_something("Four"),
      do_something("Five"),
   ]),
   do_something("Three")])

# Create another parallel structure using the above pipeline as one of its tasks
_parallel = parallel([
   _pipeline,
   do_something("Six"),
   do_something("Seven")])

# Create a funnel structure that executes all its tasks passing the result to the
```

(continues on next page)

(continued from previous page)

```
# single, final task
_funnel = funnel([
   do_something("Eight"),
   _parallel,
   do_something("Nine")])

# Gather the result from the _funnel and send it to do_something("Four")
print("FUNNEL: ", _funnel(do_something("Four")).get())
```

CHAPTER

SIX

DATA FLOWS

ElasticCode provides a unique and easy way to deploy distributed data flows (sometimes called workflows). These flows are contructed by using the ElasticCode object model and linking them together.

To review the ElasticCode object model, we have the following taxonomy used in an ElasticCode network.

- **Nodes**
  - **Agents**
    - **Workers**
      - **Processors**

        Sockets

        Tasks
        -Arguments

For a given processor, multiple sockets can be exposed that allow incoming requests to different functions (tasks) within the processors python module code. Links between outputs of one socket and inputs of another are established using Plugs.

Each Plug has a source socket and a target socket, such that when the function associated with the source socket completes, its output is used as input to the target socket function. These requests persist on a queue and execute in an orderly fashion to not stress resources. Since processors are bound to one or more CPUs, they can service requests in parallel but will only execute requests when resources are free to do so.

Because functions are coupled into data flows using loose coupling, you are able to change the topology of your data flow anytime. Execution will follow the path of the current dataflow.

When connecting a Plug to a target Socket, you can specify a specific argument for the target function that the plug is connected to.

For example, consider this target function:

```
def add_two(one, two):
    return one+two
```

diagram

It has two arguments *one* and *two* by name. You might have a data flow with two separate inputs to *add_two* where one plug satisfies the *one* argument and the other plug satisfies the *two* argument. In this design,`add_two` will only trigger once both arguments have *arrived* at the socket. This means arguments can arrive at different times and different orders.

CHAPTER

SEVEN

ARCHITECTURE

ElasticCode is a scalable, high-performance network architecture that separates concerns across layers. Each layer has best-of-breed components that manage the responsibility of that layer. The slides below show the different layers and their responsibilities, starting with the bottom-most layer.

7.1 Managed Compute

ElasticCode takes a different approach to staging and executing python code on its network. Other frameworks or libraries allow you to define your functions in your execution environment and serialize that code to remote workers for execution. Obviously that has some serious security implications in a *shared, managed compute environment*. So ElasticCode does not allow this. Rather, you request ElasticCode to mount your code through a secure git repostiory URL. This becomes *the contract* between you and ElasticCode and allows ElasticCode to securely load your code into its network.

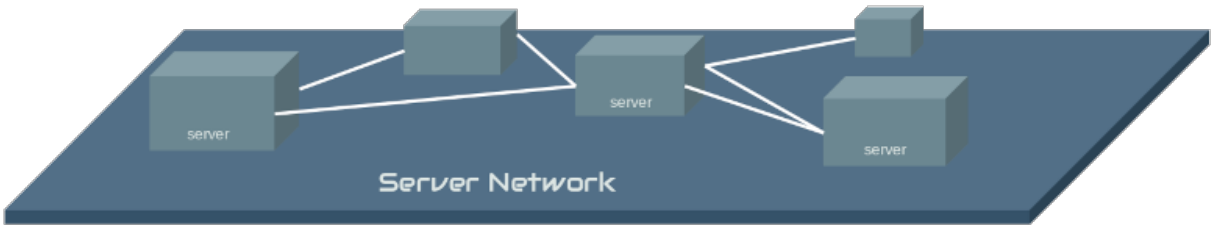
This approach also allows administrators to control white and blacklists for what repositories of code it trusts.

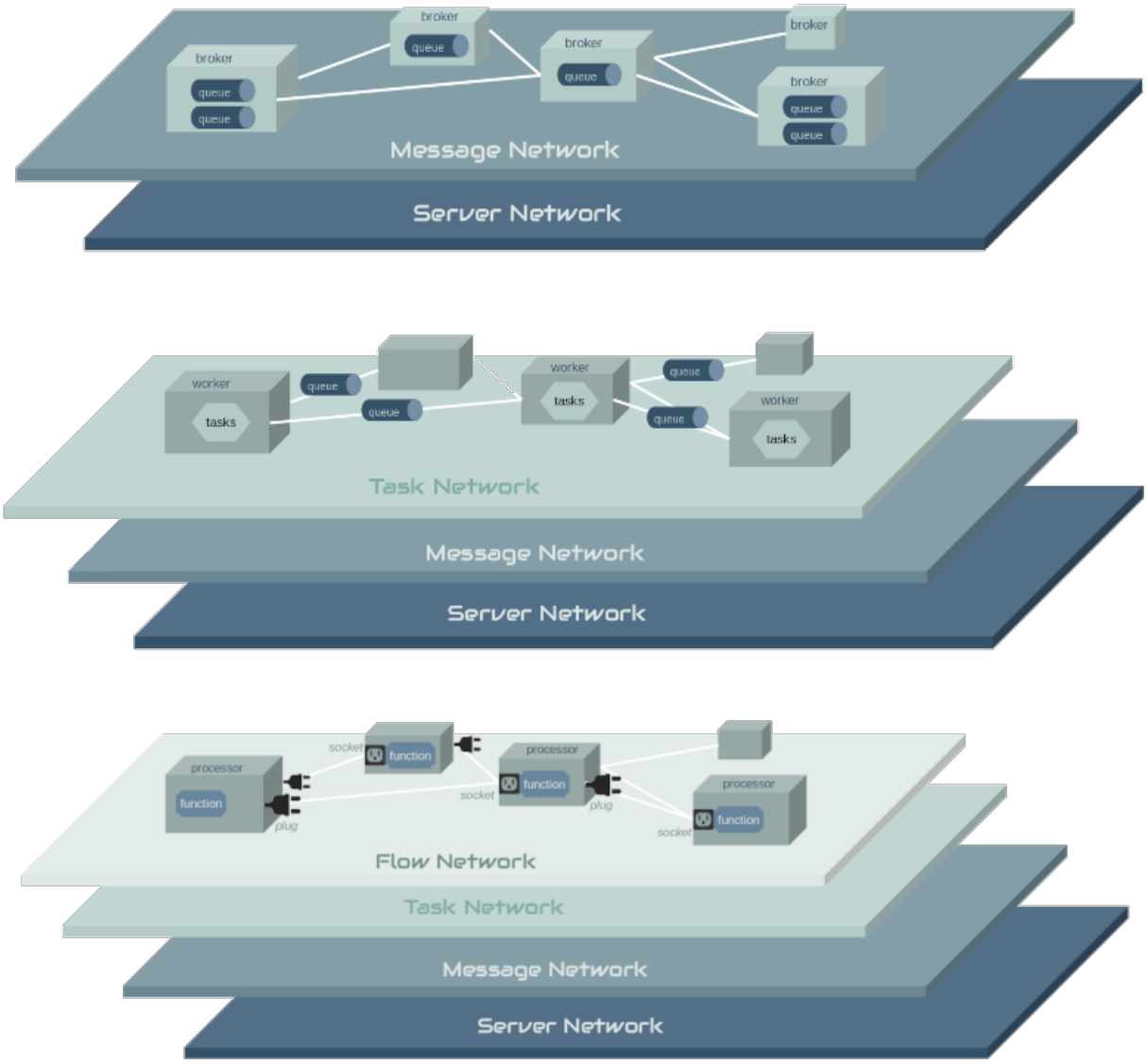
7.2 Code Isolation

Each ElasticCode worker that mounts a git repository, will create a virtual environment for that code and execute the repositories *setup.py* to install the code in that virtual environment. This is beneficial for a number of reasons, but most importantly it keeps the environment for the mounted code separate from the ElasticCode agent's python environment.

7.3 Layered Design

ElasticCode is a distributed, scalable architecture and as such it is relationship between connected hardware & service layers interacting as a whole.



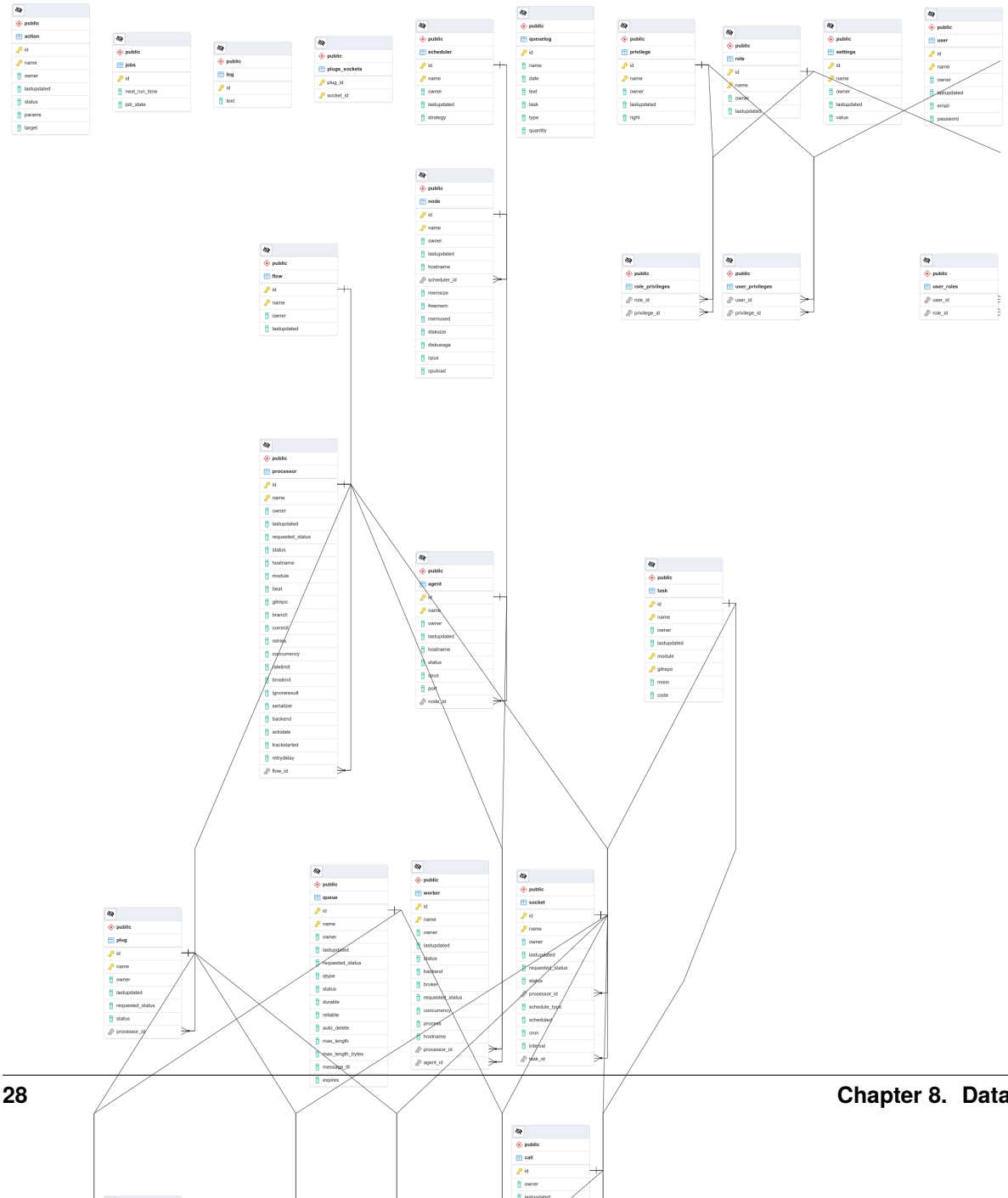

broker
queue
broker
queue
queue
broker
queue
broker
queue
queue
Message Network
Server Network
worker
tasks
queue
queue
worker
tasks
queue
queue
worker
tasks
Task Network
Message Network
Server Network
processor
function
plug
socket
function
socket
processor
function
plug
socket
processor
function
Flow Network
Task Network
Message Network
Server Network

CHAPTER

EIGHT

DATABASE

8.1 Data Model



CHAPTER

NINE

SERVERS

9.1 Web

9.2 API

CHAPTER

TEN

CLI

10.1 Examples

Listing 1: The 'flow' command is the single command for building and managing a ElasticCode network.

```
$ flow
Usage: flow [OPTIONS] COMMAND [ARGS]...

  CLI for creating & managing flow networks

Options:
  --debug               Debug switch
  -d, --db TEXT         Database URI
  --backend TEXT        Task queue backend
  --broker TEXT         Message broker URI
  -a, --api TEXT        Message broker API URI
  -u, --user TEXT       Message broker API user
  -p, --password TEXT   Message broker API password
  -i, --ini TEXT        flow .ini configuration file
  -c, --config          Configure pyfi
  --help                Show this message and exit.

Commands:
  add         Add an object to the database
  agent       Commands for remote agent management
  api         API server admin
  compose     Manage declarative infrastructure files
  db          Database operations
  delete      Delete an object from the database
  listen      Listen to a processor output
  login       Log into flow CLI
  logout      Logout current user
  ls          List database objects and their relations
  network     Network operations
  node        Node management operations
  proc        Run or manage processors
  scheduler   Scheduler management commands
  task        Pyfi task management
  update      Update a database object
```

(continues on next page)

(continued from previous page)

```
  user       User commands
  web        Web server admin
  whoami     Database login user
  worker     Run pyfi worker
```

10.1.1 Database

Listing 2: Flow database sub-commands

```
$ flow db
Usage: flow db [OPTIONS] COMMAND [ARGS]...

  Database operations

Options:
  --help  Show this message and exit.

Commands:
  drop     Drop all database tables
  init     Initialize database tables
  json     Dump the database to JSON
  migrate  Perform database migration/upgrade
  rebuild  Drop and rebuild database tables
```

10.1.2 Objects

There are numerous objects within an ElasticCode network. Some are infrastructure related, others are service related. Using the ElasticCode CLI you create, update and manage these objects in the database, which acts as a **single source of truth** for the entire ElasticCode network. All the deployed ElasticCode services (e.g. agents) *react* to changes in the ElasticCode database. So you could say that ElasticCode is *reactive* on a distributed, network-scale.

Some of the system objects and CLI commands are shown below.

10.1.3 Queues

Listing 3: Add a queue to the database

```
$ flow add queue --help
Usage: flow add queue [OPTIONS]

  Add queue object to the database

Options:
  -n, --name TEXT                 [required]
  -t, --type [topic|direct|fanout]
                                  [default: direct; required]
  --help                          Show this message and exit.
```

10.1.4 Processors

Listing 4: Add a processor to the database

```
$ flow add processor --help
Usage: flow add processor [OPTIONS]

  Add processor to the database

Options:
  -n, --name TEXT               Name of this processor  [required]
  -m, --module TEXT             Python module (e.g. some.module.path
                                [required]
  -h, --hostname TEXT           Target server hostname
  -w, --workers INTEGER         Number of worker tasks
  -r, --retries INTEGER         Number of retries to invoke this processor
  -g, --gitrepo TEXT            Git repo URI  [required]
  -c, --commit TEXT             Git commit id for processor code
  -rs, --requested_status TEXT  The requested status for this processor
  -b, --beat                    Enable the beat scheduler
  -br, --branch TEXT            Git branch to be used for checkouts
  -p, --password TEXT           Password to access this processor
  -rq, --requirements TEXT      requirements.txt file
  -e, --endpoint TEXT           API endpoint path
  -a, --api BOOLEAN             Has an API endpoint
  -cs, --cpus INTEGER           Number of CPUs for default deployment
  -d, --deploy                  Enable the beat scheduler
  -mp, --modulepath TEXT        Relative repo path to python module file
  --help                        Show this message and exit.
```

Listing 5: Specific processor subcommands

```
$ flow proc
Usage: flow proc [OPTIONS] COMMAND [ARGS]...

  Run or manage processors

Options:
  --id TEXT  ID of processor
  --help     Show this message and exit.

Commands:
  pause    Pause a processor
  remove   Remove a processor
  restart  Start a processor
  resume   Pause a processor
  start    Start a processor
  stop     Stop a processor
```

10.1.5 Calls

Listing 6: Call subcommands

```
$ flow ls calls --help
Usage: flow ls calls [OPTIONS]

  List calls

Options:
  -p, --page INTEGER
  -r, --rows INTEGER
  -u, --unfinished
  -a, --ascend
  -i, --id
  -t, --tracking
  -tk, --task
  --help              Show this message and exit.
```

Listing 7: flow ls calls

```
$ flow ls calls
+------+-----+--------------------------------------+--------------------------------------
↪-+----------+------------------------------+--------------------------------------+-------
↪---------------------+------------------------------+----------+
| Page | Row |                 Name                 |                  ID                  
↪ |  Owner   |        Last Updated          |                Socket                |      
↪   Started           |          Finished            |  State   |
+------+-----+--------------------------------------+--------------------------------------
↪-+----------+------------------------------+--------------------------------------+-------
↪---------------------+------------------------------+----------+
|  1   |  1  |    pyfi.processors.sample.do_this    | e3f73300-f3fd-4230-ba11-
↪258d4f5a17f4 | postgres | 2021-09-13 19:30:19.933346 |    pyfi.processors.sample.do_
↪this    | 2021-09-13 19:30:19.903573 | 2021-09-13 19:30:19.932491 | finished |
|  1   |  2  | pyfi.processors.sample.do_something | e3bf09c5-ae45-4772-b301-
↪c394acae3c4e | postgres | 2021-09-13 19:30:19.885993 | pyfi.processors.sample.do_
↪something | 2021-09-13 19:30:19.847282 | 2021-09-13 19:30:19.885440 | finished |
|  1   |  3  |    pyfi.processors.sample.do_this    | a58de16a-1b92-4acb-81c1-
↪92e81cb6ea56 | postgres | 2021-09-13 19:29:49.944219 |    pyfi.processors.sample.do_
↪this    | 2021-09-13 19:29:49.917225 | 2021-09-13 19:29:49.943415 | finished |
|  1   |  4  | pyfi.processors.sample.do_something | 58df162a-ac2e-40b7-9e27-
↪635c61a4d9a7 | postgres | 2021-09-13 19:29:49.868975 | pyfi.processors.sample.do_
↪something | 2021-09-13 19:29:49.820097 | 2021-09-13 19:29:49.868109 | finished |
|  1   |  5  |    pyfi.processors.sample.do_this    | 60d8b91d-1b8b-433c-a289-
↪5704856d37d1 | postgres | 2021-09-13 19:29:19.907705 |    pyfi.processors.sample.do_
↪this    | 2021-09-13 19:29:19.880742 | 2021-09-13 19:29:19.906931 | finished |
|  1   |  6  | pyfi.processors.sample.do_something | 66c78849-9052-48d0-ae62-
↪59942d544096 | postgres | 2021-09-13 19:29:19.861880 | pyfi.processors.sample.do_
↪something | 2021-09-13 19:29:19.824456 | 2021-09-13 19:29:19.861330 | finished |
|  1   |  7  |    pyfi.processors.sample.do_this    | e5189a71-9805-492e-a8d7-
↪e5eb2b8d68d3 | postgres | 2021-09-13 19:28:49.873301 |    pyfi.processors.sample.do_
↪this    | 2021-09-13 19:28:49.842724 | 2021-09-13 19:28:49.872176 | finished |
|  1   |  8  | pyfi.processors.sample.do_something | 35fd3635-743a-4015-acfe-
```

(continues on next page)

(continued from previous page)

```
↪c5a8f62ef65d | postgres | 2021-09-13 19:28:49.812921 | pyfi.processors.sample.do_
↪something | 2021-09-13 19:28:49.789503 | 2021-09-13 19:28:49.812406 | finished |
|  1   |  9  |    pyfi.processors.sample.do_this    | 4136ebe2-ee96-4b74-ba0e-
↪33d8c5974252 | postgres | 2021-09-13 19:28:19.830508 |    pyfi.processors.sample.do_
↪this   | 2021-09-13 19:28:19.805839 | 2021-09-13 19:28:19.829667 | finished |
|  1   |  10 | pyfi.processors.sample.do_something | 707f18c5-5708-4c70-81fb-
↪ca0afb30e28b | postgres | 2021-09-13 19:28:19.789542 | pyfi.processors.sample.do_
↪something | 2021-09-13 19:28:19.764792 | 2021-09-13 19:28:19.788999 | finished |
+------+-----+-------------------------------------+-------------------------------------
↪-+----------+-----------------------------+-------------------------------------+-------
↪---------------------+-----------------------------+----------+
Page 1 of 383 of 3830 total records
```

Listing 8: flow ls call –help

```
$ flow ls call --help
Usage: flow ls call [OPTIONS]

List details about a call record

Options:
--id TEXT        ID of call
-n, --name TEXT  Name of call
-r, --result     Include result of call
-t, --tree       Show forward call tree
-g, --graph      Show complete call graph
-f, --flow       Show all calls in a workflow
--help           Show this message and exit.
```

Listing 9: flow ls call –id e3bf09c5-ae45-4772-b301-c394acae3c4e

```
$ flow ls call --id e3bf09c5-ae45-4772-b301-c394acae3c4e
+-------------------------------------+-------------------------------------+----------
↪+-----------------------------+-------------------------------------+-------------------
↪---------+-----------------------------+----------+
|                 Name                |                  ID                 |  Owner  ␣
↪|        Last Updated         |               Socket                |          Started ␣
↪          |          Finished          |  State   |
+-------------------------------------+-------------------------------------+----------
↪+-----------------------------+-------------------------------------+-------------------
↪---------+-----------------------------+----------+
| pyfi.processors.sample.do_something | e3bf09c5-ae45-4772-b301-c394acae3c4e | postgres␣
↪| 2021-09-13 19:30:19.885993 | pyfi.processors.sample.do_something | 2021-09-13␣
↪19:30:19.847282 | 2021-09-13 19:30:19.885440 | finished |
+-------------------------------------+-------------------------------------+----------
↪+-----------------------------+-------------------------------------+-------------------
↪---------+-----------------------------+----------+
Provenance
+-------------------------------------+-------------+-------------+
|                 Task                | Task Parent | Flow Parent |
+-------------------------------------+-------------+-------------+
| a13ba1e7-78f9-4644-9c29-696dfd89e9e4 |     None    |     None    |
```

(continues on next page)

(continued from previous page)

```
+-------------------------------------+-------------+-------------+
Events
+----------+--------------------------------------+----------+--------------------------
↪-+-------------------------------------------------------+
|   Name   |                  ID                  |  Owner   |        Last Updated      ␣
↪ |                         Note                          |
+----------+--------------------------------------+----------+--------------------------
↪-+-------------------------------------------------------+
| received | 8e8845d5-cd32-40d9-93c7-e95f7500926c | postgres | 2021-09-13 19:30:19.
↪844512 |  Received task pyfi.processors.sample.do_something  |
|  prerun  | a2507cd1-1d72-4ad1-be74-375aac29f1c4 | postgres | 2021-09-13 19:30:19.
↪874789 | Prerun for task pyfi.processors.sample.do_something |
| postrun  | f8b5ff03-e0e3-467d-9257-a682f0865581 | postgres | 2021-09-13 19:30:19.
↪886504 |                  Postrun for task                   |
+----------+--------------------------------------+----------+--------------------------
↪-+-------------------------------------------------------+
```

Listing 10: flow ls call –id e3bf09c5-ae45-4772-b301-c394acae3c4e –tree

```
$ flow ls call --id e3bf09c5-ae45-4772-b301-c394acae3c4e --tree
pyfi.processors.sample.do_something
      └──────────────────────
               pyfi.processors.sample.do_this
```

10.1.6 Listening

The listen command allows you to listen to the pubsub channels associated with queues and processors. A kind of *network sniffer* that displays in real-time the various message traffic, compute results, lifecycle events etc. You can provide your own custom class to receive the results which is designed to provide a simple and loosely coupled mechanism for system integrations.

Listing 11: Messages will be displayed as they are generated within the network.

```
$ flow listen --help
Usage: flow listen [OPTIONS]

Listen to a processor output

Options:
-n, --name TEXT     Name of processor  [required]
-c, --channel TEXT  Listen channel (e.g. task, log, etc)  [required]
-a, --adaptor TEXT  Adaptor class function (e.g. my.module.class.function)
--help              Show this message and exit.
$ flow listen --name pyfi.queue1.proc1 --channel task
Listening to pyfi.queue1.proc1
{'type': 'psubscribe', 'pattern': None, 'channel': b'pyfi.queue1.proc1.task', 'data': 1}
{'type': 'pmessage', 'pattern': b'pyfi.queue1.proc1.task', 'channel': b'pyfi.queue1.
↪proc1.task', 'data': b'{"channel": "task", "state": "received", "date": "2021-09-13␣
↪19:37:20.094443", "room": "pyfi.queue1.proc1"}'}
```

(continues on next page)

(continued from previous page)

```
{'type': 'pmessage', 'pattern': b'pyfi.queue1.proc1.task', 'channel': b'pyfi.queue1.
↪proc1.task', 'data': b'{"channel": "task", "state": "running", "date": "2021-09-13␣
↪19:37:20.108668", "room": "pyfi.queue1.proc1"}'}
{'type': 'pmessage', 'pattern': b'pyfi.queue1.proc1.task', 'channel': b'pyfi.queue1.
↪proc1.task', 'data': b'{"module": "pyfi.processors.sample", "date": "2021-09-13␣
↪19:37:20.133327", "resultkey": "celery-task-meta-b3feb181-484d-4b98-aba8-daabd07ee3d1",
↪ "message": "{\\"module\\": \\"pyfi.processors.sample\\", \\"date\\": \\"2021-09-13␣
↪19:37:20.133327\\", \\"resultkey\\": \\"celery-task-meta-b3feb181-484d-4b98-aba8-
↪daabd07ee3d1\\", \\"message\\": \\"\\\\\\"\\\\\\\\\\\\\\\\\\\\\\\\\\"Message:Hello World!\\\\\\\\\\\\
↪\\\\\"\\\\\\"\\", \\"channel\\": \\"task\\", \\"room\\": \\"pyfi.queue1.proc1\\", \\
↪"task\\": \\"do_something\\"}", "channel": "task", "room": "pyfi.queue1.proc1", "task
↪": "do_something", "state": "postrun"}'}
```

10.1.7 Running an Agent

Listing 12: FLOW agent subcommand

```
$ flow agent
Usage: flow agent [OPTIONS] COMMAND [ARGS]...

Run flow agent

Options:
--help  Show this message and exit.

Commands:
start  Run pyfi agent server
```

Listing 13: Flow agent subcommand

```
$ flow agent start --help
Usage: flow agent start [OPTIONS]

  Start an agent

Options:
  -p, --port INTEGER          Healthcheck port
  --clean                     Remove work directories before launch
  -b, --backend TEXT          Message backend URI
  -r, --broker TEXT           Message broker URI
  -n, --name TEXT             Hostname for this agent to use
  -c, --config TEXT           Config module.object import (e.g.
                              path.to.module.MyConfigClass
  -q, --queues                Run the queue monitor only
  -u, --user TEXT             Run the worker as user
  -po, --pool INTEGER         Process pool for message dispatches
  -cp, --cpus INTEGER         Number of CPUs
  -s, --size INTEGER          Maximum number of messages on worker internal
                              queue
  -h, --host TEXT             Remote hostname to start the agent via ssh
```

(continues on next page)

(continued from previous page)

```
  -wp, --workerport INTEGER  Healthcheck port for worker
  --help                     Show this message and exit.
```

10.1.8 Roles & Users

Listing 14: FLOW user, role and privilege subcommands

```
$ flow add user --help
Usage: flow add user [OPTIONS]

Add user object to the database

Options:
-n, --name TEXT      [required]
-e, --email TEXT     [required]
-p, --password TEXT  [required]
--help               Show this message and exit.

$ flow add role --help
Usage: flow add role [OPTIONS]

Add role object to the database

Options:
-n, --name TEXT  [required]
--help           Show this message and exit.

$ flow add privilege --help
Usage: flow add privilege [OPTIONS]

Add privilege to the database

Options:
-u, --user TEXT
-n, --name TEXT  [required]
-r, --role TEXT
--help           Show this message and exit.
```

Listing 15: Creating a user

```
$ flow add user
Name: joe
Email: joe@xyz
Password: 12345
CREATE USER joe WITH PASSWORD '12345'
User "joe" added
```

Listing 16: Creating a role

```
$ flow add role -n developer
bc15ee9d-a208-43a9-82d2-bf0810dc4380:developer:2021-09-15 21:50:40.714192
```

Listing 17: Adding a privilege to a user

```
$ flow add privilege -u joe -n ADD_PROCESSOR
Privilege added
```

Listing 18: List a user with role_privileges

```
$ flow ls user -n joe
+------+--------------------------------------+----------+---------+
| Name |                  ID                  |  Owner   |  Email  |
+------+--------------------------------------+----------+---------+
| joe  | a8dcf9bb-c821-4d44-82f5-828dceb4cb23 | postgres | joe@xyz |
+------+--------------------------------------+----------+---------+
Privileges
+------+---------------+----------------------------+----------+
| Name |     Right     |        Last Updated        |    By    |
+------+---------------+----------------------------+----------+
| joe  | ADD_PROCESSOR | 2021-09-15 21:46:48.611286 | postgres |
+------+---------------+----------------------------+----------+
```

Listing 19: Adding a privilege to a role

```
$ flow add privilege -r developer -n ADD_PROCESSOR
Privilege added
```

Listing 20: Adding a role to a user

```

```

10.1.9 Privileges & Rights

A **right** is an atomic string that names a particular **privilege**. It only becomes a privilege when it's associated with a user. When it's just **a name** we call it a *right*.

Listing 21: Available Rights

```
rights = ['ALL',
    'CREATE',
    'READ',
    'UPDATE',
    'DELETE',

    'DB_DROP',
    'DB_INIT',

    'START_AGENT',

    'RUN_TASK',
    'CANCEL_TASK',

    'START_PROCESSOR',
```

(continues on next page)

(continued from previous page)

```
    'STOP_PROCESSOR',
    'PAUSE_PROCESSOR',
    'RESUME_PROCESSOR',
    'LOCK_PROCESSOR',
    'UNLOCK_PROCESSOR',
    'VIEW_PROCESSOR',
    'VIEW_PROCESSOR_CONFIG',
    'VIEW_PROCESSOR_CODE',
    'EDIT_PROCESSOR_CONFIG',
    'EDIT_PROCESSOR_CODE'

    'LS_PROCESSORS',
    'LS_USERS',
    'LS_USER',
    'LS_PLUGS',
    'LS_SOCKETS',
    'LS_QUEUES',
    'LS_AGENTS',
    'LS_NODES',
    'LS_SCHEDULERS',
    'LS_WORKERS',

    'ADD_PROCESSOR',
    'ADD_AGENT',
    'ADD_NODE',
    'ADD_PLUG',
    'ADD_PRIVILEGE',
    'ADD_QUEUE',
    'ADD_ROLE',
    'ADD_SCHEDULER',
    'ADD_SOCKET',
    'ADD_USER',

    'UPDATE_PROCESSOR',
    'UPDATE_AGENT',
    'UPDATE_NODE',
    'UPDATE_PLUG',
    'UPDATE_PRIVILEGE',
    'UPDATE_QUEUE',
    'UPDATE_ROLE',
    'UPDATE_SCHEDULER',
    'UPDATE_SOCKET',
    'UPDATE_USER',

    'DELETE_PROCESSOR',
    'DELETE_AGENT',
    'DELETE_NODE',
    'DELETE_PLUG',
    'DELETE_PRIVILEGE',
    'DELETE_QUEUE',
    'DELETE_ROLE',
    'DELETE_SCHEDULER',
```

(continues on next page)

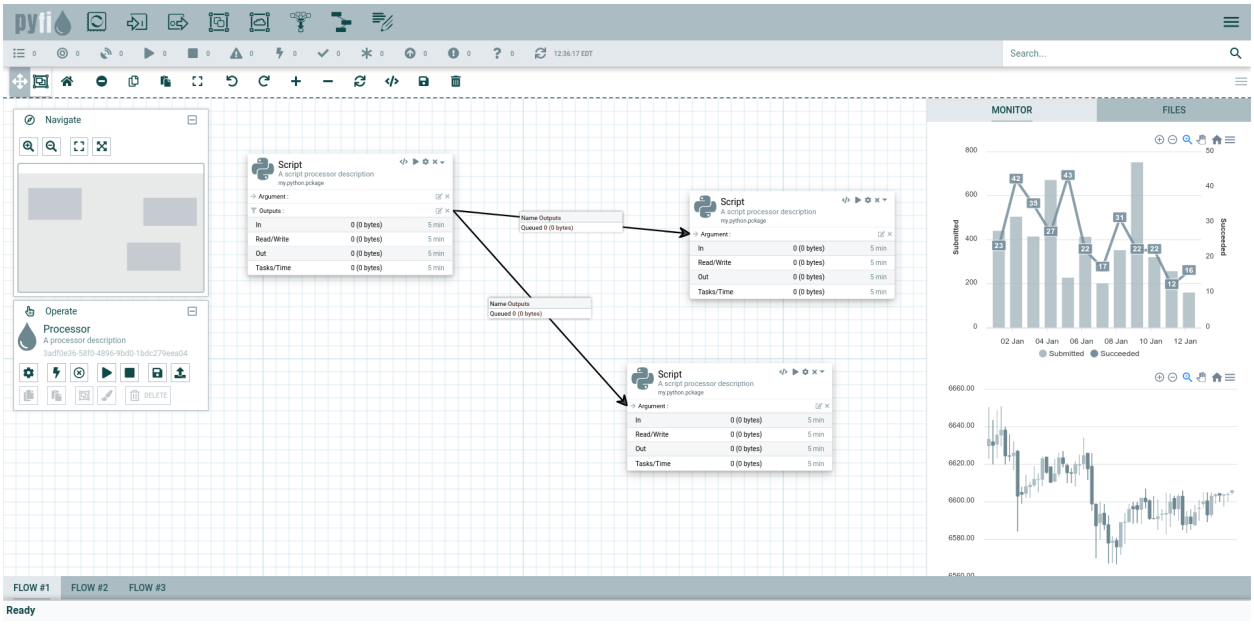
(continued from previous page)

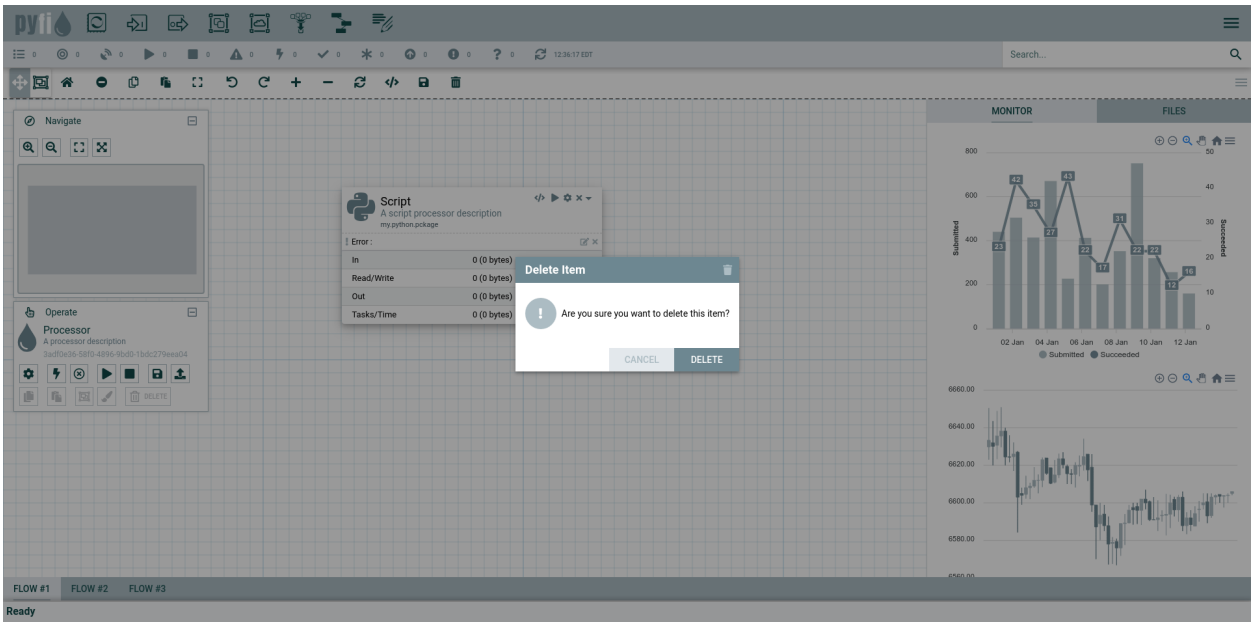
```
    'DELETE_SOCKET',
    'DELETE_USER',

    'READ_PROCESSOR',
    'READ_AGENT',
    'READ_NODE',
    'READ_PLUG',
    'READ_PRIVILEGE',
    'READ_QUEUE',
    'READ_ROLE',
    'READ_SCHEDULER',
    'READ_SOCKET',
    'READ_USER'
   ]
```

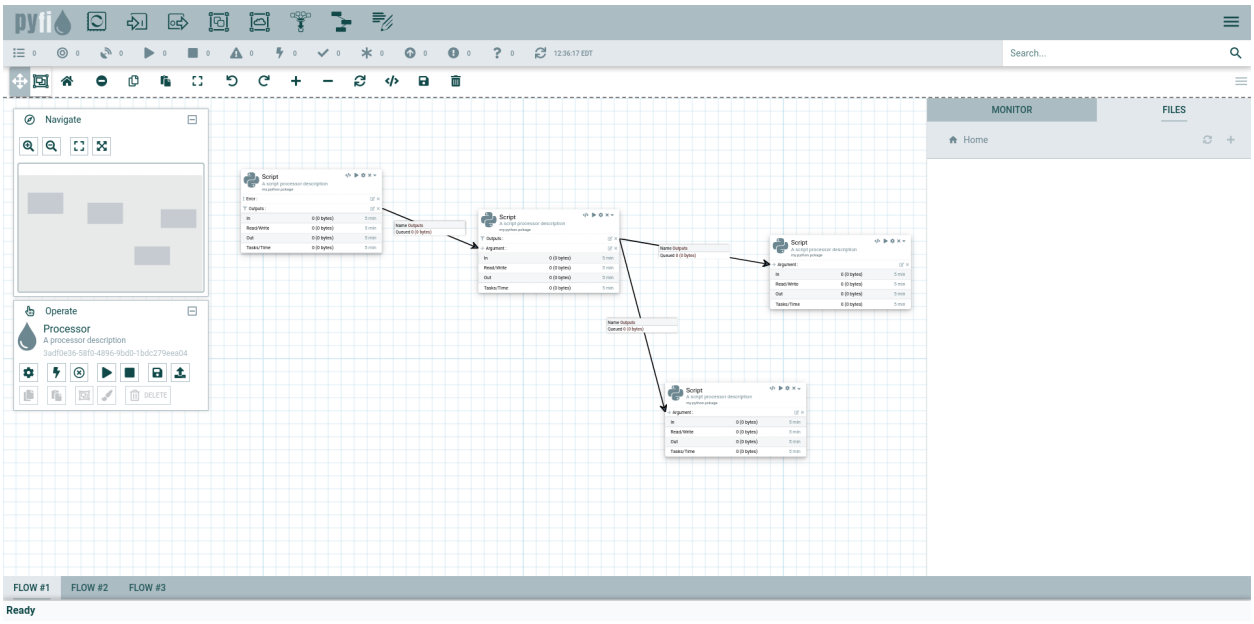
CHAPTER

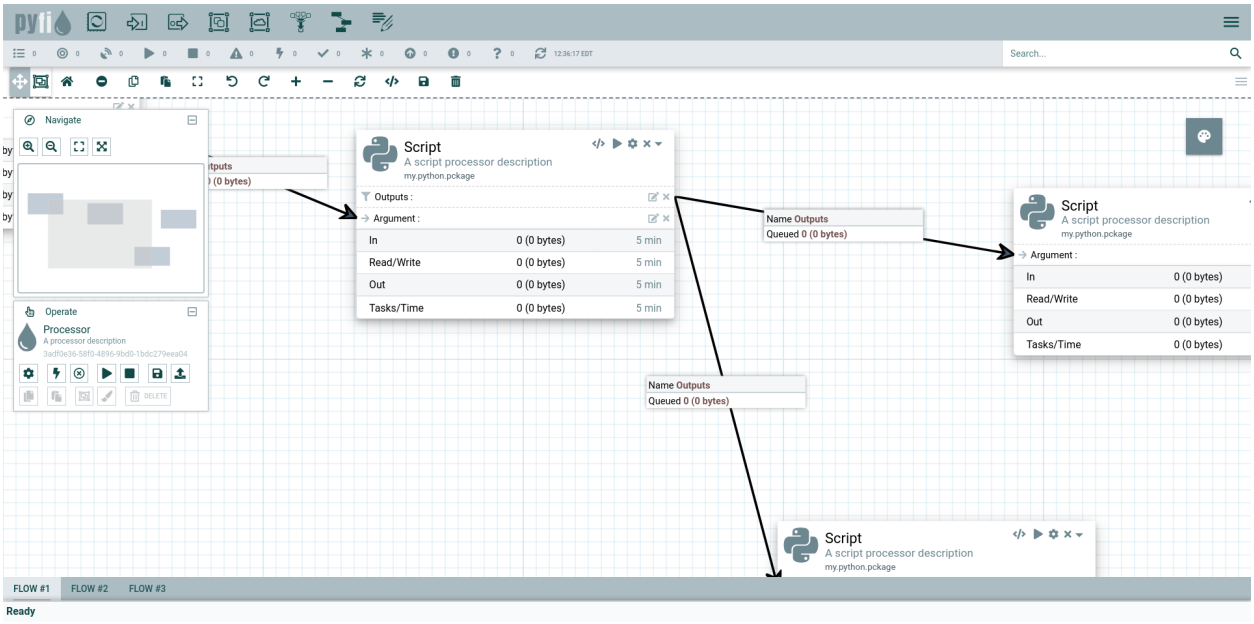
ELEVEN

UI



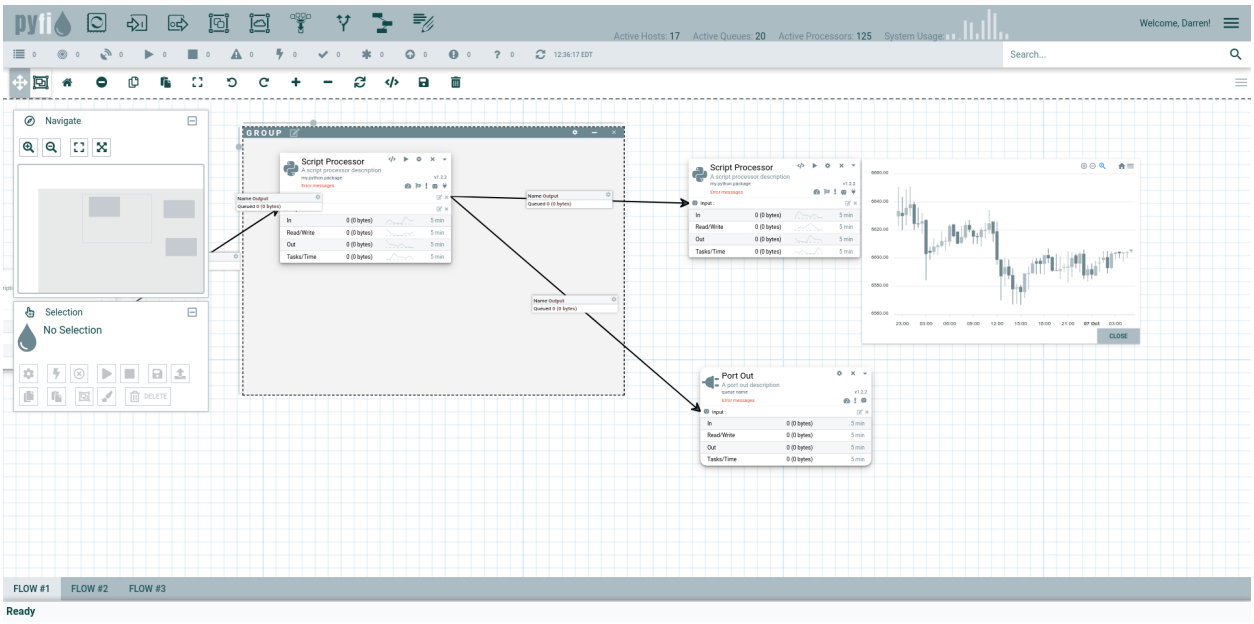


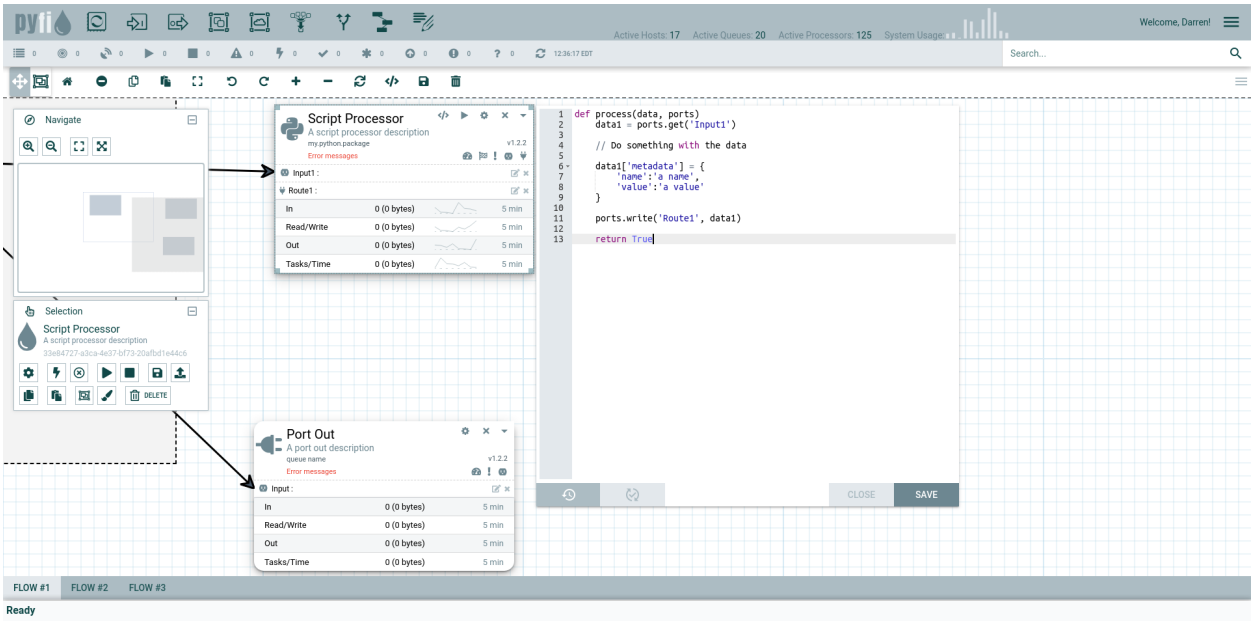

Navigate
Operate
Processor
A processor description
MONITOR
FILES
Home
FLOW #1
FLOW #2
FLOW #3
Ready

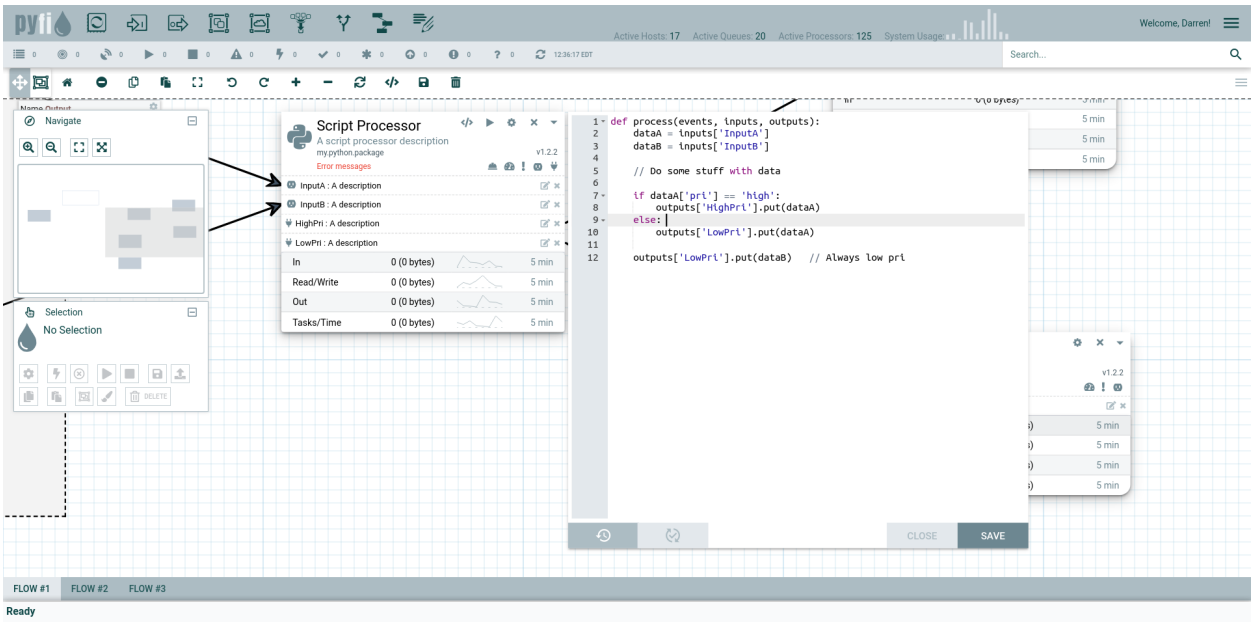

Navigate
Operate
Processor
A processor description
Script
A script processor description
my.python.pckage
Outputs :
Argument :
In 0 (0 bytes) 5 min
Read/Write 0 (0 bytes) 5 min
Out 0 (0 bytes) 5 min
Tasks/Time 0 (0 bytes) 5 min
Name Outputs
Queued 0 (0 bytes)
FLOW #1
FLOW #2
FLOW #3
Ready

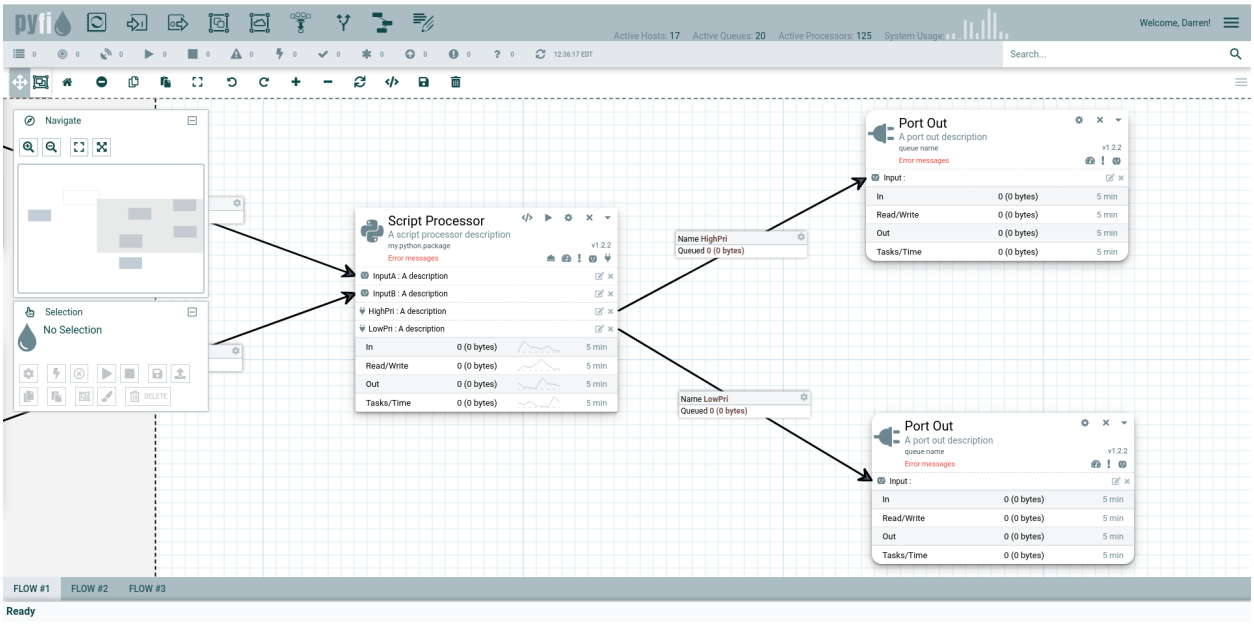

Name Outputs
Queued 0 (0 bytes)
Script
A script proces
my.python.pckage
Outputs :
Argument :
In
Read/Write
Out
Tasks/Time
Refresh
Disable
View Provenance Data
View Status History
View State
View Usage
View Connections
Center in View
Change Color
Create Template
Copy
Delete
Name Outputs
Queued 0 (0 bytes)
Script
A script processor desc
my.python.pckage
Argument :
In
Read/Write
Out
Tasks/Time
Script
A script processor description
my.python.pckage
Argument :
In 0 (0 bytes) 5 min
Read/Write 0 (0 bytes) 5 min
Out 0 (0 bytes) 5 min
Tasks/Time 0 (0 bytes) 5 min

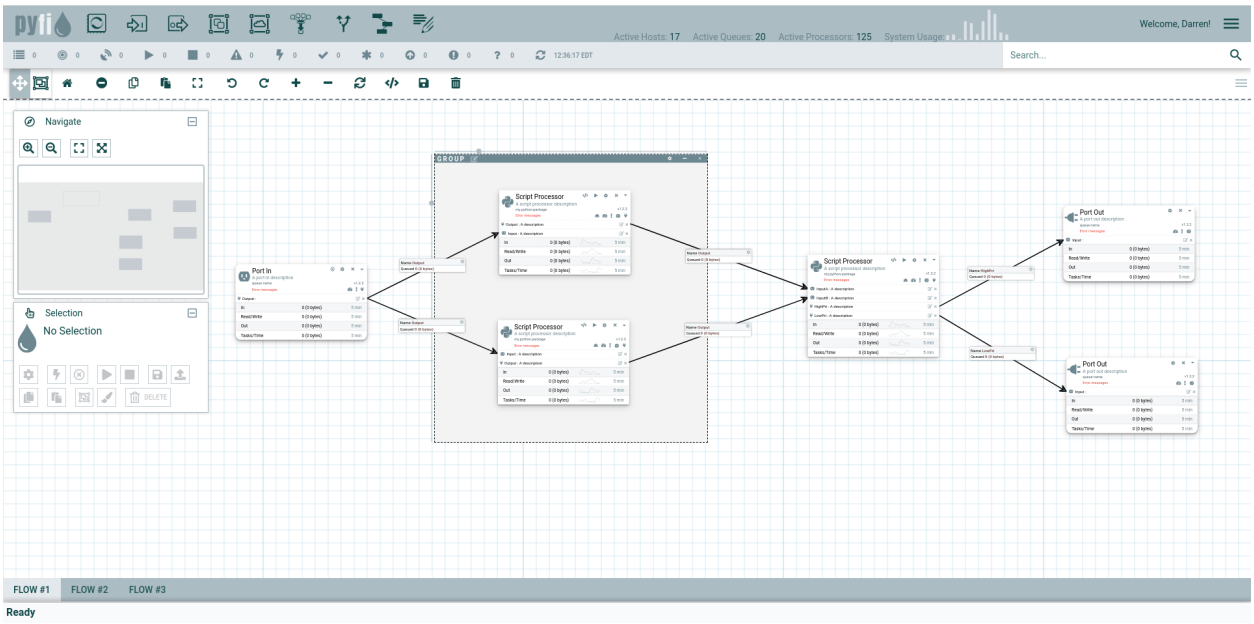
Search...
Navigate
Operate
Processor
A processor description
Script
A script processor description
my.python.pckage
In
Read/Write
Out
Tasks/Time
Configure
Disable
View Provenance Data
View Status History
View State
View Usage
View Connections
Center in View
Change Color
Group
Create Template
Copy
Delete
MONITOR
FILES
Submitted
Succeeded
02 Jan
04 Jan
06 Jan
08 Jan
10 Jan
12 Jan
FLOW #1
FLOW #2
FLOW #3
Ready


Welcome, Darren!
Navigate
Selection
No Selection
Script Processor
Port Out
FLOW #1
FLOW #2
FLOW #3
Ready


Welcome, Darren!
Navigate
Selection
Script Processor
A script processor description
Port Out
A port out description
def process(data, ports)
data1 = ports.get('Input1')
// Do something with the data
data1['metadata'] = {
'name':'a name',
'value':'a value'
}
ports.write('Route1', data1)
return True
CLOSE
SAVE
FLOW #1
FLOW #2
FLOW #3
Ready


Welcome, Darren!
Active Hosts: 17
Active Queues: 20
Active Processors: 125
System Usage:
Search...
Navigate
Selection
No Selection
Script Processor
A script processor description
my.python.package
v1.2.2
Error messages
InputA : A description
InputB : A description
HighPri : A description
LowPri : A description
In 0 (0 bytes) 5 min
Read/Write 0 (0 bytes) 5 min
Out 0 (0 bytes) 5 min
Tasks/Time 0 (0 bytes) 5 min
def process(events, inputs, outputs):
dataA = inputs['InputA']
dataB = inputs['InputB']
// Do some stuff with data
if dataA['prl'] == 'high':
outputs['HighPri'].put(dataA)
else:
outputs['LowPri'].put(dataA)
outputs['LowPri'].put(dataB) // Always low prl
CLOSE
SAVE
FLOW #1
FLOW #2
FLOW #3
Ready


Welcome, Darren!
Active Hosts: 17
Active Queues: 20
Active Processors: 125
System Usage:
Search...
Navigate
Selection
No Selection
Script Processor
A script processor description
my.python.package
v1.2.2
Error messages
InputA : A description
InputB : A description
HighPri : A description
LowPri : A description
In 0 (0 bytes) 5 min
Read/Write 0 (0 bytes) 5 min
Out 0 (0 bytes) 5 min
Tasks/Time 0 (0 bytes) 5 min
Name HighPri
Queued 0 (0 bytes)
Name LowPri
Queued 0 (0 bytes)
Port Out
A port out description
queue name
Input :
Port Out
A port out description
queue name
Input :
FLOW #1
FLOW #2
FLOW #3
Ready


Active Hosts: 17
Active Queues: 20
Active Processors: 125
System Usage:
Welcome, Darren!
Search...
Navigate
Selection
No Selection
DELETE
GROUP
Port In
Script Processor
Script Processor
Script Processor
Port Out
Port Out
FLOW #1
FLOW #2
FLOW #3
Ready

CHAPTER

TWELVE

API

12.1 CLI

See section on *CLI*

12.2 Python

12.2.1 Decorators

Listing 1: ElasticCode Python decorator API “”” Decorator API for Flow. Defines network from plain old classes and methods. “”” import os

```
from pyfi.client.api import ProcessorBase
from pyfi.client.decorators import plug, processor, socket


@processor(
    name="proc2",
    gitrepo=os.environ["GIT_REPO"],
    module="pyfi.processors.sample",
    concurrency=1,
)
class ProcessorB(ProcessorBase):
    """Description"""

    @socket(
        name="proc2.do_this",
        processor="proc2",
        arguments=True,
        queue={"name": "sockq2"},
    )
    def do_this(message):
        from random import randrange

        print("Do this!", message)
        message = "Do this String: " + str(message)
        graph = {
            "tag": {"name": "tagname", "value": "tagvalue"},
```

(continues on next page)

(continued from previous page)

```
            "name": "distance",
            "value": randrange(50),
        }
        return {"message": message, "graph": graph}


@processor(
    name="proc1",
    gitrepo=os.environ["GIT_REPO"],
    module="pyfi.processors.sample",
    concurrency=7,
)
class ProcessorA(ProcessorBase):
    """Description"""

    def get_message(self):
        return "Self message!"

    @plug(
        name="plug1",
        target="proc2.do_this",  # Must be defined above already (prevents cycles)
        queue={
            "name": "queue1",
            "message_ttl": 300000,
            "durable": True,
            "expires": 200,
        },
    )
    @socket(
        name="proc1.do_something",
        processor="proc1",
        beat=False,
        interval=15,
        queue={"name": "sockq1"},
    )
    def do_something(message):
        """do_something"""
        from random import randrange

        message = "TEXT:" + str(message)
        graph = {
            "tag": {"name": "tagname", "value": "tagvalue"},
            "name": "temperature",
            "value": randrange(10),
        }
        return {"message": message, "graph": graph}


@processor(
    name="proc3",
    gitrepo=os.environ["GIT_REPO"],
    module="pyfi.processors.sample",
```

(continues on next page)

(continued from previous page)

```
    concurrency=1,
)
class ProcessorC(ProcessorBase):
    """Description"""

    def get_message(self):
        return "Self message!"

    @plug(
        name="plug2",
        target="proc2.do_this",  # Must be defined above already (prevents cycles)
        queue={
            "name": "queue2",
            "message_ttl": 300000,
            "durable": True,
            "expires": 200,
        },
    )
    @socket(
        name="proc3.do_something",
        processor="proc3",
        beat=False,
        interval=5,
        queue={"name": "sockq3"},
    )
    def do_something(message):
        """do_something"""
        from random import randrange

        message = "TEXT2:" + str(message)
        graph = {
            "tag": {"name": "tagname", "value": "tagvalue"},
            "name": "temperature",
            "value": randrange(10),
        }
        return {"message": message, "graph": graph}


if __name__ == "__main__":
    print("Network created.")
```

12.2.2 Objects

Listing 2: ElasticCode Python Object API import json

```
from pyfi.client.api import Plug, Processor, Socket
from pyfi.client.user import USER
from pyfi.db.model import AlchemyEncoder

# Log in a user first
```

(continues on next page)

(continued from previous page)

```
  print("USER", USER)
  # Create a processor
  processor = Processor(
      name="proc1",
      beat=True,
      user=USER,
      module="pyfi.processors.sample",
      branch="main",
      concurrency=6,
      gitrepo="https://user:key@github.com/radiantone/pyfi-processors#egg=pyfi-processor
↪",
  )

  processor2 = Processor(
      name="proc2",
      user=USER,
      module="pyfi.processors.sample",
      hostname="agent1",
      concurrency=6,
      branch="main",
      gitrepo="https://user:key@github.com/radiantone/pyfi-processors#egg=pyfi-processor
↪",
  )

  processor3 = Processor(
      name="proc3",
      user=USER,
      module="pyfi.processors.sample",
      hostname="agent2",
      concurrency=6,
      branch="main",
      gitrepo="https://user:pword@github.com/radiantone/pyfi-processors#egg=pyfi-
↪processor",
  )

  # Create a socket on the processor to receive requests for the do_something python␣
↪function(task)
  do_something = Socket(
      name="pyfi.processors.sample.do_something",
      user=USER,
      interval=5,
      processor=processor,
      queue={"name": "pyfi.queue1"},
      task="do_something",
  )

  print(json.dumps(do_something.socket, indent=4, cls=AlchemyEncoder))
  # Create a socket on the processor to receive requests for the do_this python␣
↪function(task)
  do_this = Socket(
      name="pyfi.processors.sample.do_this",
      user=USER,
```

(continues on next page)

(continued from previous page)

```
    processor=processor2,
    queue={"name": "pyfi.queue2"},
    task="do_this",
)
do_this2 = Socket(
    name="pyfi.processors.sample.do_this",
    user=USER,
    processor=processor3,
    queue={"name": "pyfi.queue3"},
    task="do_this",
)

do_something2 = Socket(
    name="proc2.do_something",
    user=USER,
    processor=processor2,
    queue={"name": "pyfi.queue1"},
    task="do_something",
)

# Create a plug that connects one processor to a socket of another
plug = Plug(
    name="plug1",
    processor=processor,
    user=USER,
    source=do_something,
    queue={"name": "pyfi.queue3"},
    target=do_this,
)
```

12.2.3 Lambda

Listing 3: ElasticCode Python Lambda API

```
from pyfi.client.api import funnel, parallel, pipeline
from pyfi.client.example.api import do_something_p as do_something
from pyfi.client.example.api import do_this_p as do_this

"""
An example app on top of pyfi. References existing infrastructure and then runs␣
↪complex workflows and parallel operations on it
"""
_pipeline = pipeline(
    [
        do_something("One"),
        do_something("Two"),
        parallel(
            [
                do_this("Four"),
                do_this("Five"),
```

(continues on next page)

(continued from previous page)

```
            ]
        ),
        do_this("Three"),
    ]
)
print(_pipeline().get())
_parallel = parallel([_pipeline, do_something("Six"), do_something("Seven")])

_funnel = funnel(
    [do_something("Eight"), _parallel, do_something("Nine")], do_something("A")
)

_funnel2 = funnel([_parallel, do_something("Ten")], do_something("B"))

_funnel3 = funnel([_funnel, _funnel2])

result = _funnel3(do_something("Eleven"))
print("FUNNEL: ", result.get())
```

12.3 ORM

Listing 4: ElasticCode Python SQL Model API “”” Class database model definitions “”” import json from datetime import datetime from typing import Any, Optional

```
from oso import Oso
from sqlalchemy import (
    Boolean,
    Column,
    DateTime,
    Enum,
    Float,
    ForeignKey,
    Integer,
    LargeBinary,
    String,
    Table,
    Text,
    and_,
    literal_column,
)
from sqlalchemy.dialects.postgresql import DOUBLE_PRECISION
from sqlalchemy.ext.compiler import compiles
from sqlalchemy.ext.declarative import DeclarativeMeta, declared_attr
from sqlalchemy.orm import declarative_base, foreign, relationship
from sqlalchemy.schema import CreateColumn

Base: Any = declarative_base(name="Base")

oso = Oso()
```

(continues on next page)

(continued from previous page)

```
@compiles(CreateColumn, "postgresql")
def use_identity(element, compiler, **kw):
    text = compiler.visit_create_column(element, **kw)
    text = text.replace("SERIAL", "INT GENERATED BY DEFAULT AS IDENTITY")
    return text


class AlchemyEncoder(json.JSONEncoder):
    def default(self, obj):
        from datetime import datetime

        if isinstance(obj.__class__, DeclarativeMeta):
            # an SQLAlchemy class
            fields = {}
            for field in [
                x for x in dir(obj) if not x.startswith("_") and x != "metadata"
            ]:
                data = obj.__getattribute__(field)
                try:
                    # this will fail on non-encodable values, like other classes
                    if type(data) is datetime:
                        data = str(data)
                    json.dumps(data)
                    fields[field] = data
                except TypeError:
                    fields[field] = None
            # a json-encodable dict
            return fields

        return json.JSONEncoder.default(self, obj)


class HasLogins(object):
    @declared_attr
    def logins(cls):
        return relationship(
            "LoginModel",
            order_by="desc(LoginModel.created)",
            primaryjoin=lambda: and_(foreign(LoginModel.user_id) == cls.id),
            lazy="select",
        )


class HasLogs(object):
    @declared_attr
    def logs(cls):
        return relationship(
            "LogModel",
            order_by="desc(LogModel.created)",
            primaryjoin=lambda: and_(
```

(continues on next page)

(continued from previous page)

```
                foreign(LogModel.oid) == cls.id,
                LogModel.discriminator == cls.__name__,
            ),
            lazy="select",
        )


class BaseModel(Base):
    """
    Docstring
    """

    __abstract__ = True

    id = Column(
        String(40),
        autoincrement=False,
        default=literal_column("uuid_generate_v4()"),
        unique=True,
        primary_key=True,
    )
    name = Column(String(80), unique=True, nullable=False, primary_key=True)
    owner = Column(String(40), default=literal_column("current_user"))

    status = Column(String(20), nullable=False, default="ready")
    requested_status = Column(String(40), default="ready")

    enabled = Column(Boolean)
    created = Column(DateTime, default=datetime.now, nullable=False)
    lastupdated = Column(
        DateTime, default=datetime.now, onupdate=datetime.now, nullable=False
    )

    def __repr__(self):
        return json.dumps(self, cls=AlchemyEncoder)


class LogModel(Base):
    """
    Docstring
    """

    __tablename__ = "log"

    id = Column(
        String(40),
        autoincrement=False,
        default=literal_column("uuid_generate_v4()"),
        unique=True,
        primary_key=True,
    )
```

(continues on next page)

(continued from previous page)

```
    user_id = Column(String, ForeignKey("users.id"), nullable=False)
    user = relationship("UserModel", lazy=True)

    public = Column(Boolean, default=False)
    created = Column(DateTime, default=datetime.now, nullable=False)
    oid = Column(String(40), primary_key=True)
    discriminator = Column(String(40))
    text = Column(String(80), nullable=False)
    source = Column(String(40), nullable=False)

    def __repr__(self):
        return json.dumps(self, cls=AlchemyEncoder)


rights = [
    "ALL",
    "CREATE",
    "READ",
    "UPDATE",
    "DELETE",
    "DB_DROP",
    "DB_INIT",
    "START_AGENT",
    "RUN_TASK",
    "CANCEL_TASK",
    "START_PROCESSOR",
    "STOP_PROCESSOR",
    "PAUSE_PROCESSOR",
    "RESUME_PROCESSOR",
    "LOCK_PROCESSOR",
    "UNLOCK_PROCESSOR",
    "VIEW_PROCESSOR",
    "VIEW_PROCESSOR_CONFIG",
    "VIEW_PROCESSOR_CODE",
    "EDIT_PROCESSOR_CONFIG",
    "EDIT_PROCESSOR_CODE" "LS_PROCESSORS",
    "LS_USERS",
    "LS_USER",
    "LS_PLUGS",
    "LS_SOCKETS",
    "LS_QUEUES",
    "LS_AGENTS",
    "LS_NODES",
    "LS_SCHEDULERS",
    "LS_WORKERS",
    "ADD_PROCESSOR",
    "ADD_AGENT",
    "ADD_NODE",
    "ADD_PLUG",
    "ADD_PRIVILEGE",
    "ADD_QUEUE",
    "ADD_ROLE",
```

(continues on next page)

(continued from previous page)

```
    "ADD_SCHEDULER",
    "ADD_SOCKET",
    "ADD_USER",
    "UPDATE_PROCESSOR",
    "UPDATE_AGENT",
    "UPDATE_NODE",
    "UPDATE_PLUG",
    "UPDATE_ROLE",
    "UPDATE_SCHEDULER",
    "UPDATE_SOCKET",
    "UPDATE_USER",
    "DELETE_PROCESSOR",
    "DELETE_AGENT",
    "DELETE_NODE",
    "DELETE_PLUG",
    "DELETE_PRIVILEGE",
    "DELETE_QUEUE",
    "DELETE_ROLE",
    "DELETE_SCHEDULER",
    "DELETE_SOCKET",
    "DELETE_USER",
    "READ_PROCESSOR",
    "READ_AGENT",
    "READ_NODE",
    "READ_LOG",
    "READ_PLUG",
    "READ_PRIVILEGE",
    "READ_QUEUE",
    "READ_ROLE",
    "READ_SCHEDULER",
    "READ_SOCKET",
    "READ_USER",
]


class PrivilegeModel(BaseModel):
    """
    Docstring
    """

    __tablename__ = "privilege"

    right = Column("right", Enum(*rights, name="right"))


role_privileges = Table(
    "role_privileges",
    Base.metadata,
    Column("role_id", ForeignKey("role.id")),
    Column("privilege_id", ForeignKey("privilege.id")),
)
```

(continues on next page)

(continued from previous page)

```
class RoleModel(BaseModel):
    """
    Docstring
    """

    __tablename__ = "role"

    privileges = relationship(
        "PrivilegeModel", secondary=role_privileges, lazy="subquery"
    )


user_privileges_revoked = Table(
    "user_privileges_revoked",
    Base.metadata,
    Column("user_id", ForeignKey("users.id")),
    Column("privilege_id", ForeignKey("privilege.id")),
)

user_privileges = Table(
    "user_privileges",
    Base.metadata,
    Column("user_id", ForeignKey("users.id")),
    Column("privilege_id", ForeignKey("privilege.id")),
)

user_roles = Table(
    "user_roles",
    Base.metadata,
    Column("user_id", ForeignKey("users.id")),
    Column("role_id", ForeignKey("role.id")),
)


class UserModel(HasLogins, BaseModel):
    """
    Docstring
    """

    __tablename__ = "users"
    email = Column(String(120), unique=True, nullable=False)
    password = Column(String(60), unique=False, nullable=False)
    clear = Column(String(60), unique=False, nullable=False)

    privileges = relationship(
        "PrivilegeModel", secondary=user_privileges, lazy="subquery"
    )

    revoked = relationship(
        "PrivilegeModel", secondary=user_privileges_revoked, lazy="subquery"
    )
```

(continues on next page)

(continued from previous page)

```
    roles = relationship("RoleModel", secondary=user_roles, lazy="subquery")


socket_types = ["RESULT", "ERROR"]

plug_types = ["RESULT", "ERROR"]

schedule_types = ["CRON", "INTERVAL"]

strategies = ["BALANCED", "EFFICIENT"]


class FileModel(BaseModel):

    __tablename__ = "file"

    path = Column(String(120))
    filename = Column(String(80))
    collection = Column(String(80))
    code = Column(Text)
    type = Column(String(40))
    icon = Column(String(40))
    versions = relationship(
        "VersionModel", back_populates="file", cascade="all, delete-orphan"
    )


flows_versions = Table(
    "flows_versions",
    Base.metadata,
    Column("flow_id", ForeignKey("flow.id"), primary_key=True),
    Column("version_id", ForeignKey("versions.id"), primary_key=True),
)


class FlowModel(BaseModel):
    """
    A flow model
    """

    __tablename__ = "flow"

    # Collection of processors within this flow. A processor can reside
    # in multiple flows at once
    processors = relationship("ProcessorModel", lazy=True)

    # File reference for this flow. i.e. it's saved state
    file_id = Column(String, ForeignKey("file.id"), nullable=False)
    file = relationship(
        "FileModel", lazy=True, cascade="all, delete-orphan", single_parent=True
    )
```

(continues on next page)

(continued from previous page)

```
    # List of versions associated with this flow
    versions = relationship("VersionModel", secondary=flows_versions, lazy=True)


class AgentModel(BaseModel):
    """
    Docstring
    """

    __tablename__ = "agent"
    hostname = Column(String(60))
    cpus = Column(Integer)
    port = Column(Integer)
    pid = Column(Integer)

    workers = relationship(
        "WorkerModel", backref="agent", lazy=True, cascade="all, delete-orphan"
    )

    node_id = Column(String(40), ForeignKey("node.id"), nullable=False)


class ActionModel(BaseModel):
    """
    Docstring
    """

    __tablename__ = "action"
    params = Column(String(80))

    # host, worker, processor, queue, or all
    target = Column(String(20), nullable=False)


class WorkerModel(BaseModel):
    """
    Docstring
    """

    __tablename__ = "worker"
    backend = Column(String(40), nullable=False)
    broker = Column(String(40), nullable=False)
    concurrency = Column(Integer)
    process = Column(Integer)
    port = Column(Integer)
    hostname = Column(String(60))

    workerdir = Column(String(256))

    processor = relationship("ProcessorModel")
    processor_id = Column(
```

(continues on next page)

(continued from previous page)

```
        String(40), ForeignKey("processor.id", ondelete="CASCADE"), nullable=False
    )

    deployment_id = Column(String(40), ForeignKey("deployment.id"), nullable=True)

    deployment = relationship("DeploymentModel", back_populates="worker")

    agent_id = Column(String(40), ForeignKey("agent.id"), nullable=False)

    # agent = relationship("AgentModel", back_populates="worker")


class ContainerModel(BaseModel):
    __tablename__ = "container"

    container_id = Column(String(80), unique=True, nullable=False)


class VersionModel(Base):
    __tablename__ = "versions"

    id = Column(
        String(40),
        autoincrement=False,
        default=literal_column("uuid_generate_v4()"),
        unique=True,
        primary_key=True,
    )
    name = Column(String(80), unique=False, nullable=False)
    file_id = Column(String, ForeignKey("file.id"), nullable=False)
    file = relationship(
        "FileModel", lazy=True, cascade="all, delete-orphan", single_parent=True
    )
    owner = Column(String(40), default=literal_column("current_user"))
    flow = Column(Text, unique=False, nullable=False)

    version = Column(
        DateTime, default=datetime.now, onupdate=datetime.now, nullable=False
    )


class DeploymentModel(BaseModel):
    __tablename__ = "deployment"

    name = Column(String(80), unique=False, nullable=False)
    hostname = Column(String(80), nullable=False)
    cpus = Column(Integer, default=1, nullable=False)
    processor_id = Column(String(40), ForeignKey("processor.id"), nullable=False)

    worker = relationship(
        "WorkerModel", lazy=True, uselist=False, back_populates="deployment"
    )
```

(continues on next page)

(continued from previous page)

```
class ProcessorModel(HasLogs, BaseModel):
    """
    Docstring
    """

    __tablename__ = "processor"

    module = Column(String(80), nullable=False)
    beat = Column(Boolean)
    gitrepo = Column(String(180))
    branch = Column(String(30), default="main")
    commit = Column(String(50), nullable=True)
    gittag = Column(String(50), nullable=True)
    retries = Column(Integer)
    concurrency = Column(Integer)
    receipt = Column(String(80), nullable=True)
    ratelimit = Column(String(10), default=60)
    perworker = Column(Boolean, default=True)
    timelimit = Column(Integer)
    ignoreresult = Column(Boolean)
    serializer = Column(String(10))
    backend = Column(String(80))
    ackslate = Column(Boolean)
    trackstarted = Column(Boolean)
    disabled = Column(Boolean)
    retrydelay = Column(Integer)
    password = Column(Boolean)
    requirements = Column(Text)
    endpoint = Column(Text)
    modulepath = Column(Text)
    icon = Column(Text)
    cron = Column(Text)
    hasapi = Column(Boolean)
    uistate = Column(Text)

    description = Column(Text(), nullable=True, default="Some description")
    container_image = Column(String(60))
    container_command = Column(String(180))
    container_version = Column(String(20), default="latest")
    use_container = Column(Boolean, default=False)
    detached = Column(Boolean, default=False)

    user_id = Column(String, ForeignKey("users.id"), nullable=False)
    user = relationship("UserModel", backref="processor", lazy=True)

    flow_id = Column(String(40), ForeignKey("flow.id"), nullable=True)

    password = relationship("PasswordModel", lazy=True, viewonly=True)
    password_id = Column(String, ForeignKey("passwords.id"), nullable=True)
```

(continues on next page)

(continued from previous page)

```
    plugs = relationship(
        "PlugModel", backref="processor", lazy=True, cascade="all, delete-orphan"
    )

    deployments = relationship(
        "DeploymentModel", backref="processor", lazy=True, cascade="all, delete-orphan"
    )

    sockets = relationship(
        "SocketModel", backref="processor", lazy=True, cascade="all, delete-orphan"
    )


class JobModel(Base):
    __tablename__ = "jobs"

    id = Column(String(200), primary_key=True)
    next_run_time = Column(DOUBLE_PRECISION)
    job_state = Column(LargeBinary)


class PasswordModel(BaseModel):
    __tablename__ = "passwords"

    id = Column(
        String(40),
        autoincrement=False,
        default=literal_column("uuid_generate_v4()"),
        unique=True,
        primary_key=True,
    )
    password = Column(String(60), nullable=False)

    processor = relationship("ProcessorModel", lazy=True, uselist=False)


class NetworkModel(BaseModel):
    __tablename__ = "network"

    schedulers = relationship(
        "SchedulerModel", backref="network", lazy=True, cascade="all, delete"
    )

    queues = relationship(
        "QueueModel", backref="network", lazy=True, cascade="all, delete"
    )
    nodes = relationship(
        "NodeModel", backref="network", lazy=True, cascade="all, delete"
    )

    user_id = Column(String, ForeignKey("users.id"), nullable=False)
    user = relationship("UserModel", lazy=True)
```

(continues on next page)

(continued from previous page)

```
class WorkModel(BaseModel):
    __tablename__ = "work"

    next_run_time = Column(DOUBLE_PRECISION)
    job_state = Column(LargeBinary)

    task_id = Column(String(40), ForeignKey("task.id"))
    task = relationship("TaskModel", single_parent=True)


calls_events = Table(
    "calls_events",
    Base.metadata,
    Column("call_id", ForeignKey("call.id"), primary_key=True),
    Column("event_id", ForeignKey("event.id"), primary_key=True),
)


class CallModel(BaseModel):
    """
    Docstring
    """

    __tablename__ = "call"

    name = Column(String(80), unique=False, nullable=False)
    state = Column(String(10))
    parent = Column(String(80), nullable=True)
    taskparent = Column(String(80), nullable=True)
    resultid = Column(String(80))
    celeryid = Column(String(80))
    tracking = Column(String(80))
    argument = Column(String(40))

    task_id = Column(String(40), ForeignKey("task.id"), nullable=False)
    started = Column(DateTime, default=datetime.now, nullable=False)
    finished = Column(DateTime)

    socket_id = Column(String(40), ForeignKey("socket.id"), nullable=False)
    socket = relationship(
        "SocketModel", back_populates="call", lazy=True, uselist=False
    )

    events = relationship(
        "EventModel", secondary=calls_events, lazy=True, cascade="all, delete"
    )


class SchedulerModel(BaseModel):
    """
```

(continues on next page)

(continued from previous page)

```
    Docstring
    """

    __tablename__ = "scheduler"

    nodes = relationship("NodeModel", backref="scheduler", lazy=True)
    strategy = Column("strategy", Enum(*strategies, name="strategies"))

    network_id = Column(String(40), ForeignKey("network.id"))


class SettingsModel(BaseModel):
    """
    Docstring
    """

    __tablename__ = "settings"
    value = Column(String(80), nullable=False)


class NodeModel(BaseModel):
    """
    Docstring
    """

    __tablename__ = "node"
    hostname = Column(String(60))
    scheduler_id = Column(String(40), ForeignKey("scheduler.id"), nullable=True)

    memsize = Column(String(60), default="NaN")
    freemem = Column(String(60), default="NaN")
    memused = Column(Float, default=0)

    disksize = Column(String(60), default="NaN")
    diskusage = Column(String(60), default="NaN")
    cpus = Column(Integer, default=0)
    cpuload = Column(Float, default=0)

    network_id = Column(String(40), ForeignKey("network.id"))

    agent = relationship(
        "AgentModel", backref="node", uselist=False, cascade="all, delete-orphan"
    )


plugs_arguments = Table(
    "plugs_arguments",
    Base.metadata,
    Column("plug_id", ForeignKey("plug.id"), primary_key=True),
    Column("argument_id", ForeignKey("argument.id"), primary_key=True),
)
```

(continues on next page)

(continued from previous page)

```
class ArgumentModel(BaseModel):
    __tablename__ = "argument"

    name = Column(String(60), nullable=False)
    position = Column(Integer, default=0)
    kind = Column(Integer)

    task_id = Column(String(40), ForeignKey("task.id"))

    user_id = Column(String, ForeignKey("users.id"), nullable=False)
    user = relationship("UserModel", lazy=True)
    plugs = relationship("PlugModel", backref="argument")


class TaskModel(BaseModel):
    """
    Docstring
    """

    __tablename__ = "task"

    module = Column(String(120), nullable=False, primary_key=True)
    gitrepo = Column(String(180), nullable=False, primary_key=True)
    """
    Tasks can also be mixed-in to the module loaded by the processor as new functions
    using the code field, which must contain a function
    """
    mixin = Column(Boolean, default=False)

    source = Column(Text)  # Repo module function code
    code = Column(Text)  # Source code override for task

    sockets = relationship("SocketModel", back_populates="task")

    arguments = relationship("ArgumentModel", backref="task")


class EventModel(BaseModel):
    """
    Events are linked to call objects: received, prerun, postrun
    """

    __tablename__ = "event"
    note = Column(String(80), nullable=False)
    name = Column(String(80), nullable=False)

    call_id = Column(String(40), ForeignKey("call.id"))
    call = relationship(
        "CallModel",
        back_populates="events",
        single_parent=True,
```

(continues on next page)

(continued from previous page)

```
            cascade="all, delete-orphan",
        )


    sockets_queues = Table(
        "sockets_queues",
        Base.metadata,
        Column("socket_id", ForeignKey("socket.id")),
        Column("queue_id", ForeignKey("queue.id")),
    )

    plugs_source_sockets = Table(
        "plugs_source_sockets",
        Base.metadata,
        Column("plug_id", ForeignKey("plug.id"), primary_key=True),
        Column("socket_id", ForeignKey("socket.id"), primary_key=True),
    )
    plugs_target_sockets = Table(
        "plugs_target_sockets",
        Base.metadata,
        Column("plug_id", ForeignKey("plug.id"), primary_key=True),
        Column("socket_id", ForeignKey("socket.id"), primary_key=True),
    )


    class GateModel(BaseModel):
        __tablename__ = "gate"

        open = Column(Boolean)
        task_id = Column(String(40), ForeignKey("task.id"))


    class SocketModel(BaseModel):
        """
        Docstring
        """

        __tablename__ = "socket"
        processor_id = Column(String(40), ForeignKey("processor.id"), nullable=False)

        schedule_type = Column("schedule_type", Enum(*schedule_types, name="schedule_type
↪"))

        scheduled = Column(Boolean)
        cron = Column(String(20))

        description = Column(Text(), nullable=True, default="Some description")
        interval = Column(Integer)
        task_id = Column(String(40), ForeignKey("task.id"))
        task = relationship(
            "TaskModel",
            back_populates="sockets",
```

(continues on next page)

(continued from previous page)

```
        single_parent=True,
        cascade="delete, delete-orphan",
    )

    user_id = Column(String, ForeignKey("users.id"), nullable=False)
    user = relationship("UserModel", lazy=True)

    # Wait for all sourceplugs to deliver their data before invoking the task
    wait = Column(Boolean, default=False)

    sourceplugs = relationship("PlugModel", secondary=plugs_source_sockets)

    targetplugs = relationship("PlugModel", secondary=plugs_target_sockets)

    queue = relationship("QueueModel", secondary=sockets_queues, uselist=False)

    call = relationship(
        "CallModel", back_populates="socket", cascade="all, delete-orphan"
    )


plugs_queues = Table(
    "plugs_queues",
    Base.metadata,
    Column("plug_id", ForeignKey("plug.id")),
    Column("queue_id", ForeignKey("queue.id")),
)


class PlugModel(BaseModel):
    """
    Docstring
    """

    __tablename__ = "plug"

    type = Column("type", Enum(*plug_types, name="plug_type"), default="RESULT")

    processor_id = Column(String(40), ForeignKey("processor.id"), nullable=False)

    source = relationship(
        "SocketModel",
        back_populates="sourceplugs",
        secondary=plugs_source_sockets,
        uselist=False,
    )

    target = relationship(
        "SocketModel",
        back_populates="targetplugs",
        secondary=plugs_target_sockets,
        uselist=False,
```

(continues on next page)

(continued from previous page)

```
    )
    argument_id = Column(String, ForeignKey("argument.id"))

    user_id = Column(String, ForeignKey("users.id"), nullable=False)
    user = relationship("UserModel", lazy=True)

    description = Column(Text(), nullable=True, default="Some description")
    queue = relationship("QueueModel", secondary=plugs_queues, uselist=False)


class QueueModel(BaseModel):
    """
    Docstring
    """

    __tablename__ = "queue"
    qtype = Column(String(20), nullable=False, default="direct")
    durable = Column(Boolean, default=True)
    reliable = Column(Boolean, default=True)
    auto_delete = Column(Boolean, default=True)
    max_length = Column(Integer, default=-1)
    max_length_bytes = Column(Integer, default=-1)
    message_ttl = Column(Integer, default=3000)
    expires = Column(Integer, default=3000)

    network_id = Column(String(40), ForeignKey("network.id"))


class LoginModel(Base):
    __tablename__ = "login"

    id = Column(
        String(40),
        autoincrement=False,
        default=literal_column("uuid_generate_v4()"),
        unique=True,
        primary_key=True,
    )
    owner = Column(String(40), default=literal_column("current_user"))

    created = Column(DateTime, default=datetime.now, nullable=False)
    lastupdated = Column(
        DateTime, default=datetime.now, onupdate=datetime.now, nullable=False
    )
    login = Column(DateTime, default=datetime.now, nullable=False)
    token = Column(
        String(40),
        autoincrement=False,
        default=literal_column("uuid_generate_v4()"),
        unique=True,
        primary_key=True,
    )
```

(continues on next page)

(continued from previous page)

```
    user_id = Column(String, ForeignKey("users.id"), nullable=False)
    user = relationship("UserModel", lazy=True, overlaps="logins")
```

12.4 REST

CHAPTER

THIRTEEN

STACK

13.1 Containers

13.1.1 Docker

13.1.2 ElasticSearch

13.1.3 Flower

13.1.4 Insights

13.1.5 Kibana

13.1.6 Nginx

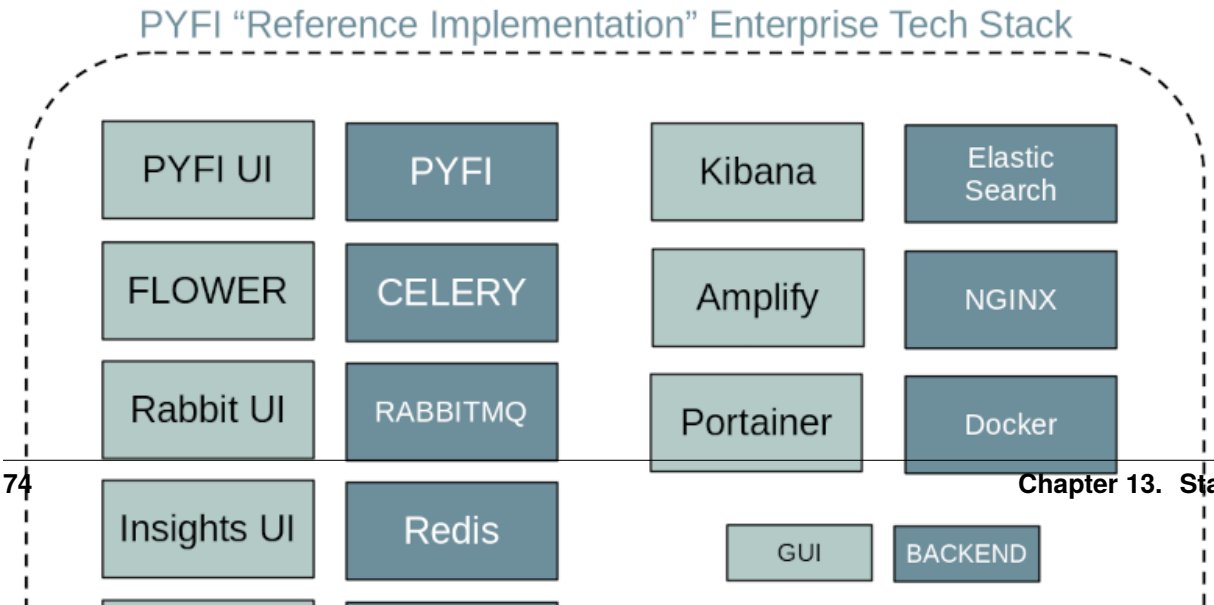
13.1.7 pgAdmin

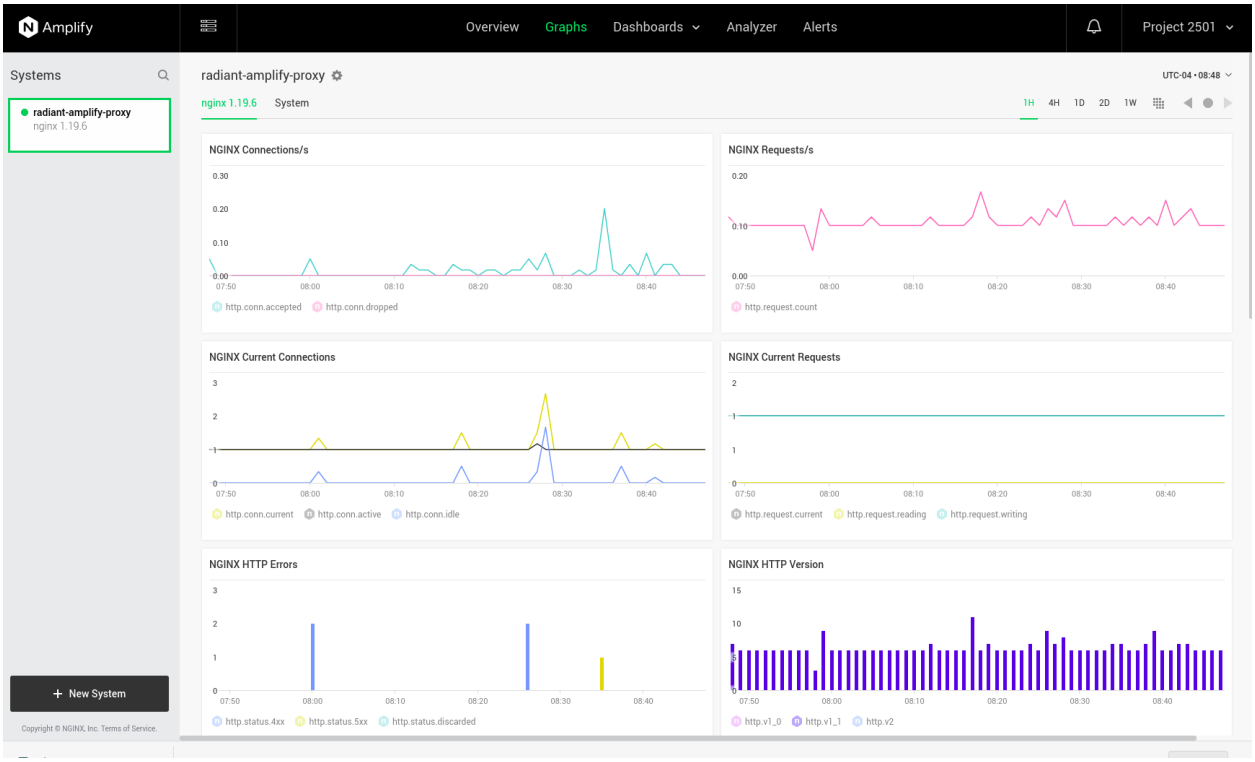
13.1.8 Portainer

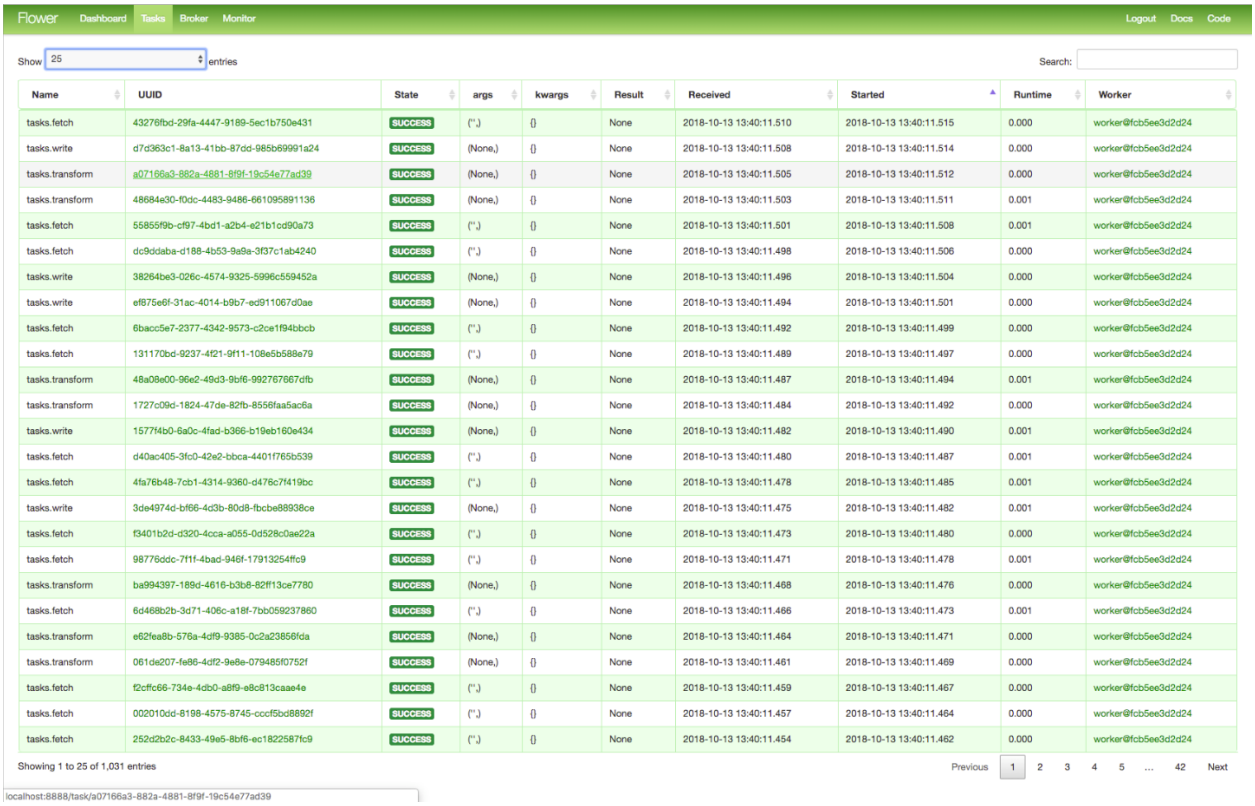
13.1.9 Postgres

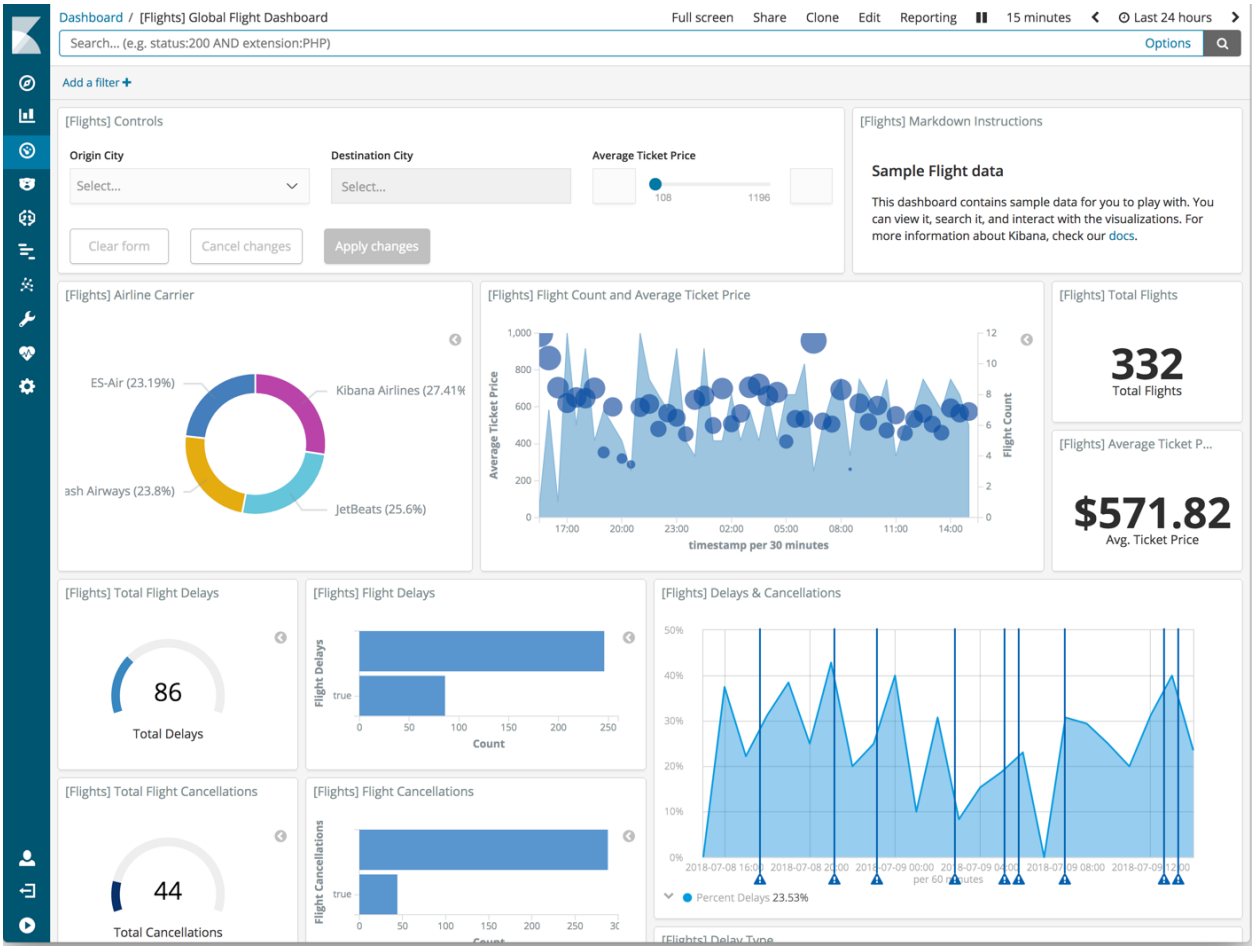
13.1.10 RabbitMQ

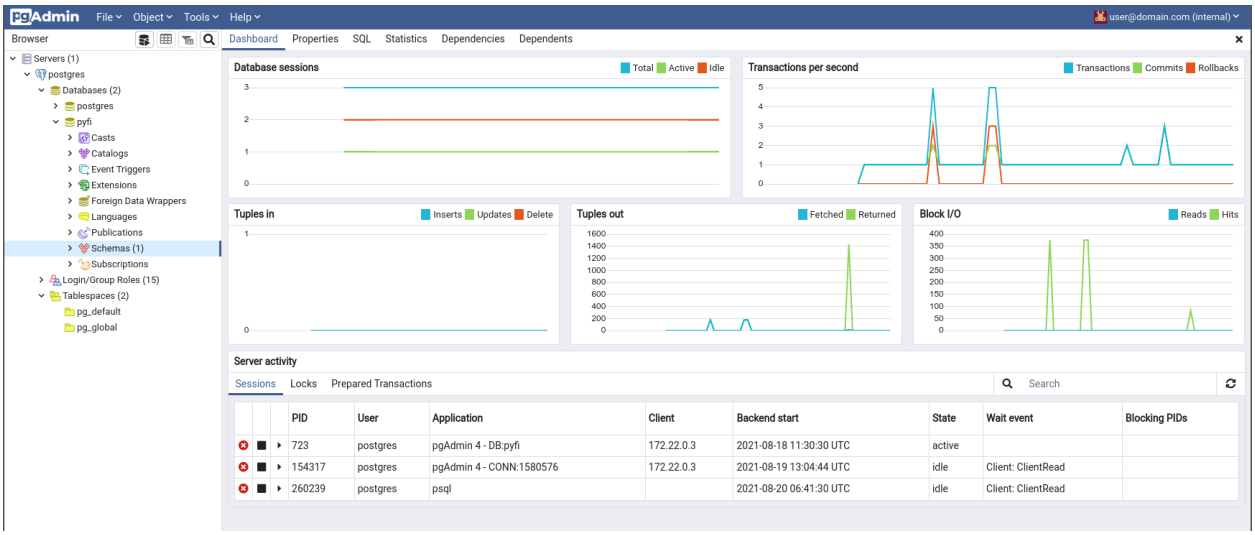
13.1.11 Redis

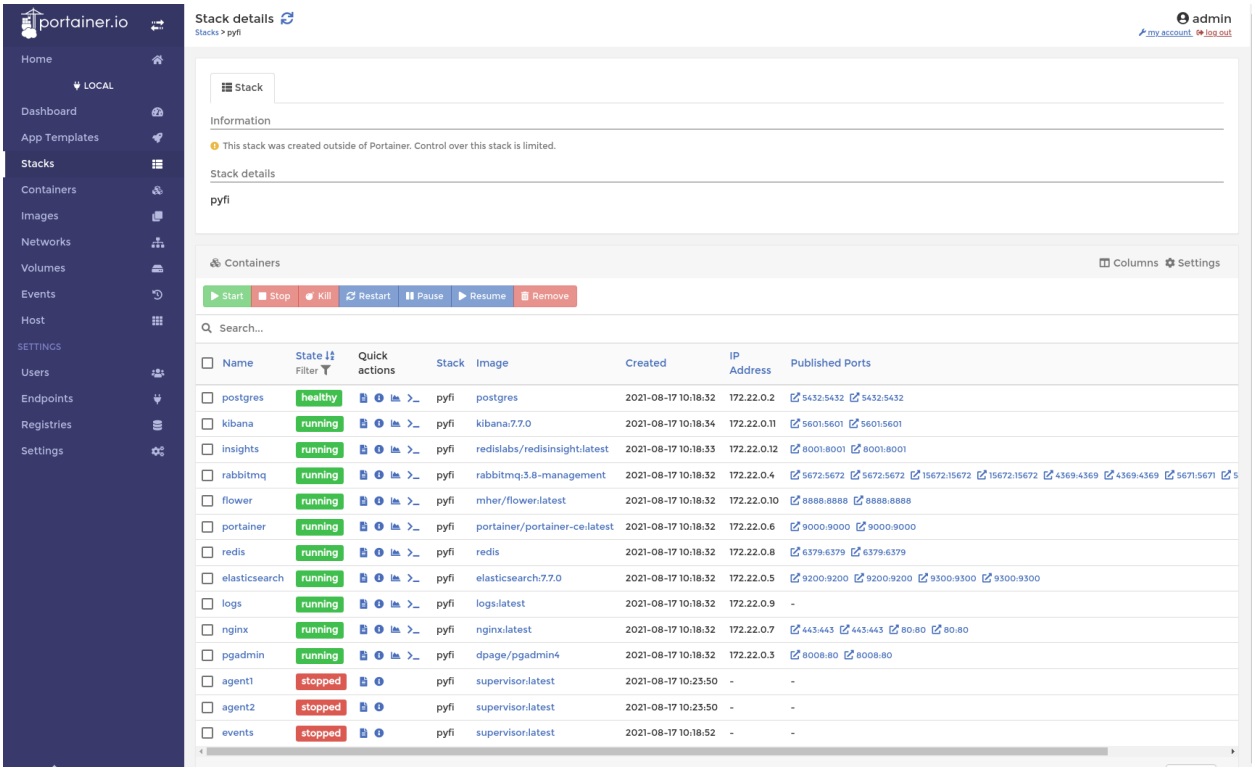


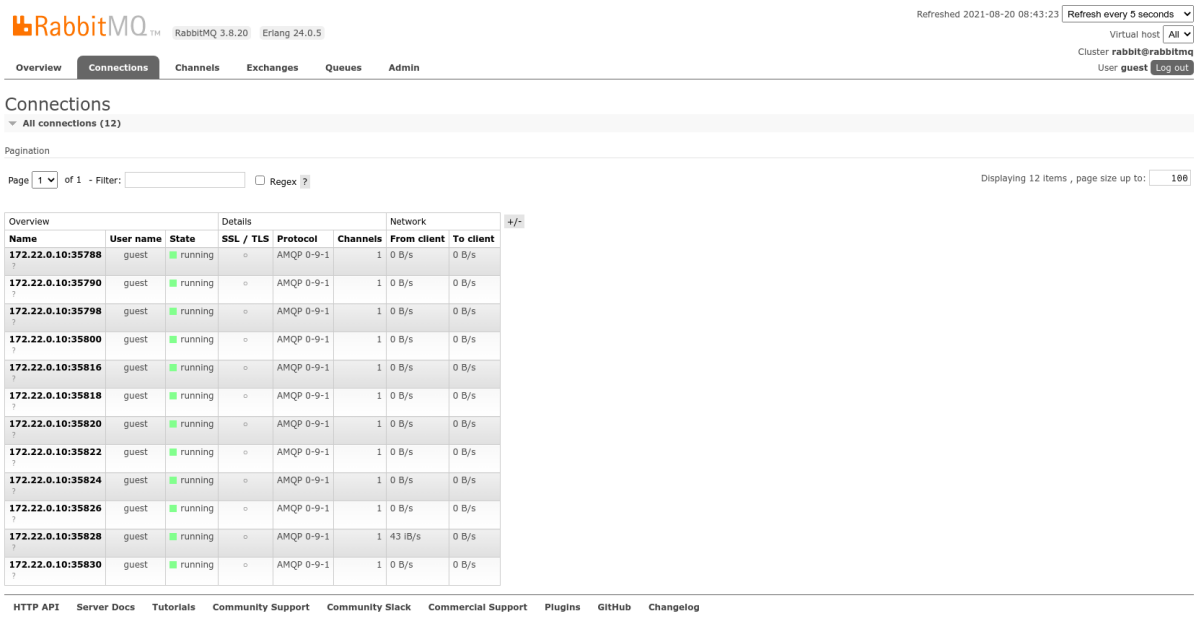

Amplify
Overview
Graphs
Dashboards
Analyzer
Alerts
Project 2501
Systems
radiant-amplify-proxy
nginx 1.19.6
radiant-amplify-proxy
nginx 1.19.6
System
UTC-04 • 08:48
1H 4H 1D 2D 1W
NGINX Connections/s
http.conn.accepted
http.conn.dropped
NGINX Requests/s
http.request.count
NGINX Current Connections
http.conn.current
http.conn.active
http.conn.idle
NGINX Current Requests
http.request.current
http.request.reading
http.request.writing
NGINX HTTP Errors
http.status.4xx
http.status.5xx
http.status.discarded
NGINX HTTP Version
http.v1_0
http.v1_1
http.v2
+ New System
Copyright © NGINX, Inc. Terms of Service.

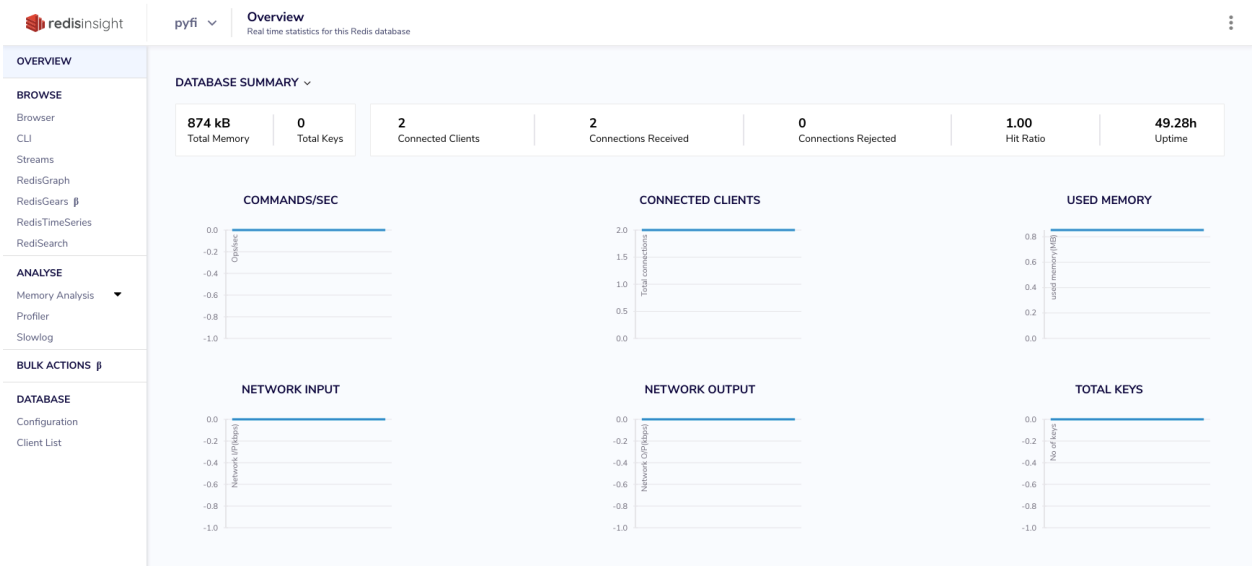

Flower
Dashboard
Tasks
Broker
Monitor
Logout
Docs
Code
Show 25 entries
Search:
Name
UUID
State
args
kwargs
Result
Received
Started
Runtime
Worker
tasks.fetch 43276fbd-29fa-4447-9189-5ec1b750e431 SUCCESS ('',) {} None 2018-10-13 13:40:11.510 2018-10-13 13:40:11.515 0.000 worker@fcb5ee3d2d24
tasks.write d7d363c1-8a13-41bb-87dd-985b69991a24 SUCCESS (None,) {} None 2018-10-13 13:40:11.508 2018-10-13 13:40:11.514 0.000 worker@fcb5ee3d2d24
tasks.transform a07166a3-882a-4881-8f9f-19c54e77ad39 SUCCESS (None,) {} None 2018-10-13 13:40:11.505 2018-10-13 13:40:11.512 0.000 worker@fcb5ee3d2d24
tasks.transform 48684e30-f0dc-4483-9486-661095891136 SUCCESS (None,) {} None 2018-10-13 13:40:11.503 2018-10-13 13:40:11.511 0.001 worker@fcb5ee3d2d24
tasks.fetch 55855f9b-cf97-4bd1-a2b4-e21b1cd90a73 SUCCESS ('',) {} None 2018-10-13 13:40:11.501 2018-10-13 13:40:11.508 0.001 worker@fcb5ee3d2d24
tasks.fetch dc9ddaba-d188-4b53-9a9a-3f37c1ab4240 SUCCESS ('',) {} None 2018-10-13 13:40:11.498 2018-10-13 13:40:11.506 0.000 worker@fcb5ee3d2d24
tasks.write 38264be3-026c-4574-9325-5996c559452a SUCCESS (None,) {} None 2018-10-13 13:40:11.496 2018-10-13 13:40:11.504 0.000 worker@fcb5ee3d2d24
tasks.write ef875e6f-31ac-4014-b9b7-ed911067d0ae SUCCESS (None,) {} None 2018-10-13 13:40:11.494 2018-10-13 13:40:11.501 0.000 worker@fcb5ee3d2d24
tasks.fetch 6bacc5e7-2377-4342-9573-c2ce1f94bbcb SUCCESS ('',) {} None 2018-10-13 13:40:11.492 2018-10-13 13:40:11.499 0.000 worker@fcb5ee3d2d24
tasks.fetch 13117 0bd-9237-4f21-9f11-108e5b588e79 SUCCESS ('',) {} None 2018-10-13 13:40:11.489 2018-10-13 13:40:11.497 0.000 worker@fcb5ee3d2d24
tasks.transform 48a08e00-96e2-49d3-9bf6-992767667dfb SUCCESS (None,) {} None 2018-10-13 13:40:11.487 2018-10-13 13:40:11.494 0.001 worker@fcb5ee3d2d24
tasks.transform 1727c09d-1824-47de-82fb-8556faa5ac6a SUCCESS (None,) {} None 2018-10-13 13:40:11.484 2018-10-13 13:40:11.492 0.000 worker@fcb5ee3d2d24
tasks.write 1577f4b0-6a0c-4fad-b366-b19eb160e434 SUCCESS (None,) {} None 2018-10-13 13:40:11.482 2018-10-13 13:40:11.490 0.001 worker@fcb5ee3d2d24
tasks.fetch d40ac405-3fc0-42e2-bbca-4401f765b539 SUCCESS ('',) {} None 2018-10-13 13:40:11.480 2018-10-13 13:40:11.487 0.001 worker@fcb5ee3d2d24
tasks.fetch 4fa76b48-7cb1-4314-9360-d476c7f419bc SUCCESS ('',) {} None 2018-10-13 13:40:11.478 2018-10-13 13:40:11.485 0.001 worker@fcb5ee3d2d24
tasks.write 3de4974d-bf66-4d3b-80d8-fbcbe88938ce SUCCESS (None,) {} None 2018-10-13 13:40:11.475 2018-10-13 13:40:11.482 0.001 worker@fcb5ee3d2d24
tasks.fetch f3401b2d-d320-4cca-a055-0d528c0ae22a SUCCESS ('',) {} None 2018-10-13 13:40:11.473 2018-10-13 13:40:11.480 0.000 worker@fcb5ee3d2d24
tasks.fetch 98776ddc-7f1f-4bad-946f-17913254ffc9 SUCCESS ('',) {} None 2018-10-13 13:40:11.471 2018-10-13 13:40:11.478 0.001 worker@fcb5ee3d2d24
tasks.transform ba994397-189d-4616-b3b8-82ff13ce7780 SUCCESS (None,) {} None 2018-10-13 13:40:11.468 2018-10-13 13:40:11.476 0.000 worker@fcb5ee3d2d24
tasks.fetch 6d468b2b-3d71-406c-a18f-7bb059237860 SUCCESS ('',) {} None 2018-10-13 13:40:11.466 2018-10-13 13:40:11.473 0.001 worker@fcb5ee3d2d24
tasks.transform e62fea8b-576a-4df9-9385-0c2a23856fda SUCCESS (None,) {} None 2018-10-13 13:40:11.464 2018-10-13 13:40:11.471 0.000 worker@fcb5ee3d2d24
tasks.transform 061de207-fe86-4df2-9e8e-079485f0752f SUCCESS (None,) {} None 2018-10-13 13:40:11.461 2018-10-13 13:40:11.469 0.000 worker@fcb5ee3d2d24
tasks.fetch f2cffc66-734e-4db0-a8f9-e8c813caae4e SUCCESS ('',) {} None 2018-10-13 13:40:11.459 2018-10-13 13:40:11.467 0.000 worker@fcb5ee3d2d24
tasks.fetch 002010dd-8198-4575-8745-cccf5bd8892f SUCCESS ('',) {} None 2018-10-13 13:40:11.457 2018-10-13 13:40:11.464 0.000 worker@fcb5ee3d2d24
tasks.fetch 252d2b2c-8433-49e5-8bf6-ec1822587fc9 SUCCESS ('',) {} None 2018-10-13 13:40:11.454 2018-10-13 13:40:11.462 0.000 worker@fcb5ee3d2d24
Showing 1 to 25 of 1,031 entries
Previous 1 2 3 4 5 ... 42 Next
localhost:8888/task/a07166a3-882a-4881-8f9f-19c54e77ad39


Dashboard / [Flights] Global Flight Dashboard
Full screen
Share
Clone
Edit
Reporting
15 minutes
Last 24 hours
Search... (e.g. status:200 AND extension:PHP)
Options
Add a filter +
[Flights] Controls
Origin City
Destination City
Average Ticket Price
Select...
Select...
108
1196
Clear form
Cancel changes
Apply changes
[Flights] Markdown Instructions
Sample Flight data
This dashboard contains sample data for you to play with. You can view it, search it, and interact with the visualizations. For more information about Kibana, check our docs.
[Flights] Airline Carrier
ES-Air (23.19%)
Kibana Airlines (27.41%
ash Airways (23.8%)
JetBeats (25.6%)
[Flights] Flight Count and Average Ticket Price
Average Ticket Price
Flight Count
timestamp per 30 minutes
[Flights] Total Flights
332
Total Flights
[Flights] Average Ticket P...
$571.82
Avg. Ticket Price
[Flights] Total Flight Delays
86
Total Delays
[Flights] Flight Delays
Flight Delays
true
Count
[Flights] Delays & Cancellations
Percent Delays 23.53%
[Flights] Total Flight Cancellations
44
Total Cancellations
[Flights] Flight Cancellations
Flight Cancellations
true


pgAdmin
File
Object
Tools
Help
user@domain.com (internal)
Browser
Dashboard
Properties
SQL
Statistics
Dependencies
Dependents
Servers (1)
postgres
Databases (2)
postgres
pyfi
Casts
Catalogs
Event Triggers
Extensions
Foreign Data Wrappers
Languages
Publications
Schemas (1)
Subscriptions
Login/Group Roles (15)
Tablespaces (2)
pg_default
pg_global
Database sessions
Total
Active
Idle
Transactions per second
Transactions
Commits
Rollbacks
Tuples in
Inserts
Updates
Delete
Tuples out
Fetched
Returned
Block I/O
Reads
Hits
Server activity
Sessions
Locks
Prepared Transactions
Search
PID
User
Application
Client
Backend start
State
Wait event
Blocking PIDs
723
postgres
pgAdmin 4 - DB:pyfi
172.22.0.3
2021-08-18 11:30:30 UTC
active
154317
postgres
pgAdmin 4 - CONN:1580576
172.22.0.3
2021-08-19 13:04:44 UTC
idle
Client: ClientRead
260239
postgres
psql
2021-08-20 06:41:30 UTC
idle
Client: ClientRead


portainer.io
Home
LOCAL
Dashboard
App Templates
Stacks
Containers
Images
Networks
Volumes
Events
Host
SETTINGS
Users
Endpoints
Registries
Settings
Stack details
Stacks > pyfi
admin
my account
log out
Stack
Information
This stack was created outside of Portainer. Control over this stack is limited.
Stack details
pyfi
Containers
Columns
Settings
Start
Stop
Kill
Restart
Pause
Resume
Remove
Search...
Name
State
Filter
Quick actions
Stack
Image
Created
IP Address
Published Ports
postgres healthy pyfi postgres 2021-08-17 10:18:32 172.22.0.2 5432:5432 5432:5432
kibana running pyfi kibana:7.7.0 2021-08-17 10:18:34 172.22.0.11 5601:5601 5601:5601
insights running pyfi redislabs/redisinsight:latest 2021-08-17 10:18:33 172.22.0.12 8001:8001 8001:8001
rabbitmq running pyfi rabbitmq:3.8-management 2021-08-17 10:18:32 172.22.0.4 5672:5672 5672:5672 15672:15672 15672:15672 4369:4369 4369:4369 5671:5671
flower running pyfi mher/flower:latest 2021-08-17 10:18:32 172.22.0.10 8888:8888 8888:8888
portainer running pyfi portainer/portainer-ce:latest 2021-08-17 10:18:32 172.22.0.6 9000:9000 9000:9000
redis running pyfi redis 2021-08-17 10:18:32 172.22.0.8 6379:6379 6379:6379
elasticsearch running pyfi elasticsearch:7.7.0 2021-08-17 10:18:32 172.22.0.5 9200:9200 9200:9200 9300:9300 9300:9300
logs running pyfi logs:latest 2021-08-17 10:18:32 172.22.0.9 -
nginx running pyfi nginx:latest 2021-08-17 10:18:32 172.22.0.7 443:443 443:443 80:80 80:80
pgadmin running pyfi dpage/pgadmin4 2021-08-17 10:18:32 172.22.0.3 8008:80 8008:80
agent1 stopped pyfi supervisor:latest 2021-08-17 10:23:50 - -
agent2 stopped pyfi supervisor:latest 2021-08-17 10:23:50 - -
events stopped pyfi supervisor:latest 2021-08-17 10:18:52 - -


RabbitMQ
RabbitMQ 3.8.20
Erlang 24.0.5
Refreshed 2021-08-20 08:43:23
Refresh every 5 seconds
Virtual host All
Cluster rabbit@rabbitmq
User guest
Log out
Overview
Connections
Channels
Exchanges
Queues
Admin
Connections
All connections (12)
Pagination
Page 1 of 1 - Filter:
Regex ?
Displaying 12 items , page size up to: 100
Overview
Details
Network
+/-
Name
User name
State
SSL / TLS
Protocol
Channels
From client
To client
172.22.0.10:35788 guest running AMQP 0-9-1 1 0 B/s 0 B/s
172.22.0.10:35790 guest running AMQP 0-9-1 1 0 B/s 0 B/s
172.22.0.10:35798 guest running AMQP 0-9-1 1 0 B/s 0 B/s
172.22.0.10:35800 guest running AMQP 0-9-1 1 0 B/s 0 B/s
172.22.0.10:35816 guest running AMQP 0-9-1 1 0 B/s 0 B/s
172.22.0.10:35818 guest running AMQP 0-9-1 1 0 B/s 0 B/s
172.22.0.10:35820 guest running AMQP 0-9-1 1 0 B/s 0 B/s
172.22.0.10:35822 guest running AMQP 0-9-1 1 0 B/s 0 B/s
172.22.0.10:35824 guest running AMQP 0-9-1 1 0 B/s 0 B/s
172.22.0.10:35826 guest running AMQP 0-9-1 1 0 B/s 0 B/s
172.22.0.10:35828 guest running AMQP 0-9-1 1 43 iB/s 0 B/s
172.22.0.10:35830 guest running AMQP 0-9-1 1 0 B/s 0 B/s
HTTP API
Server Docs
Tutorials
Community Support
Community Slack
Commercial Support
Plugins
GitHub
Changelog


redisinsight
pyfi
Overview
Real time statistics for this Redis database
OVERVIEW
BROWSE
Browser
CLI
Streams
RedisGraph
RedisGears β
RedisTimeSeries
RediSearch
ANALYSE
Memory Analysis
Profiler
Slowlog
BULK ACTIONS β
DATABASE
Configuration
Client List
DATABASE SUMMARY
874 kB
Total Memory
0
Total Keys
2
Connected Clients
2
Connections Received
0
Connections Rejected
1.00
Hit Ratio
49.28h
Uptime
COMMANDS/SEC
CONNECTED CLIENTS
USED MEMORY
NETWORK INPUT
NETWORK OUTPUT
TOTAL KEYS

CHAPTER

FOURTEEN

TUTORIALS

14.1 Examples

CHAPTER

FIFTEEN

DISCORD

CHAPTER

SIXTEEN

INDICES AND TABLES

- genindex
- modindex
- search