# Serverless Performance Simulator

***Release 0.2.2***

**Nima Mahmoudi**

**Feb 18, 2021**

# CONTENTS:

This is a project done in PACS Lab aiming to develop a performance simulator for serverless computing platforms. Using this simulator, we can calculate Quality of Service (QoS) metrics like average response time, the average probability of cold start, average running servers (directly reflecting average cost), a histogram of different events, distribution of the number of servers throughout time, and many other characteristics.

The developed performance model can be used to debug/improve analytical performance models, try new and improved management schema, or dig up a whole lot of properties of a common modern scale-per-request serverless platform.

You can check out the source code in our Github Repository. You can find the paper with details of the simultor in PACS lab website. You can use the following bibtex entry for citing our work:

```
@inproceedings{mahmoudi2021simfaas,
  author={Mahmoudi, Nima and Khazaei, Hamzeh},
  title={{SimFaaS: A Performance Simulator for Serverless Computing Platforms}},
  year={2021},
  publisher = {Springer},
  booktitle={{International Conference on Cloud Computing and Services Science}},
  pages={1--11},
  numpages = {11},
  keywords = {performance modelling, serverless computing, serverless, simulator,␣
↪performance},
  series = {CLOSER '21},
  url_paper={},
  url_pdf={https://pacs.eecs.yorku.ca/pubs/pdf/SimFaaS_CLOSER2021_Website_Preprint.
↪pdf}
}

@misc{mahmoudi2021simfaaspre,
  title={{SimFaaS: A Performance Simulator for Serverless Computing Platforms}},
  author={Nima Mahmoudi and Hamzeh Khazaei},
  year={2021},
  eprint={2102.08904},
  archivePrefix={arXiv},
  primaryClass={cs.DC},
  url_paper={https://arxiv.org/abs/2102.08904}
}
```

CHAPTER

ONE

# INSTALLATION

Install using pip:

```
pip install simfaas
```

Upgrading using pip:

```
pip install simfaas --upgrade
```

For installation in development mode:

```
git clone https://github.com/pacslab/simfaas
cd simfaas
pip install -e .
```

And in case you want to be able to execute the examples:

```
pip install -r examples/requirements.txt
```

CHAPTER

TWO

# SERVERLESS SIMULATOR

**class** simfaas.ServerlessSimulator.**ServerlessSimulator**(*arrival_process=None*, *warm_service_process=None*, *cold_service_process=None*, *expiration_threshold=600*, *max_time=86400*, *maximum_concurrency=1000*, ***kwargs*)

Bases: `object`

ServerlessSimulator is responsible for executing simulations of a sample serverless computing platform, mainly for the performance analysis and performance model evaluation purposes.

**Parameters**

- **arrival_process** (simfaas.SimProcess.SimProcess, *optional*) – The process used for generating inter-arrival samples, if absent, *arrival_rate* should be passed to signal exponential distribution, by default None
- **warm_service_process** (simfaas.SimProcess.SimProcess, *optional*) – The process which will be used to calculate service times, if absent, *warm_service_rate* should be passed to signal exponential distribution, by default None
- **cold_service_process** (simfaas.SimProcess.SimProcess, *optional*) – The process which will be used to calculate service times, if absent, *cold_service_rate* should be passed to signal exponential distribution, by default None
- **expiration_threshold** (*float, optional*) – The period of time after which the instance will be expired and the capacity release for use by others, by default 600
- **max_time** (*float, optional*) – The maximum amount of time for which the simulation should continue, by default 24*60*60 (24 hours)
- **maximum_concurrency** (*int, optional*) – The maximum number of concurrently executing function instances allowed on the system This will be used to determine when a rejection of request should happen due to lack of capacity, by default 1000

**Raises**

- **Exception** – Raises if neither arrival_process nor arrival_rate are present
- **Exception** – Raises if neither warm_service_process nor warm_service_rate are present
- **Exception** – Raises if neither cold_service_process nor cold_service_rate are present
- **ValueError** – Raises if warm_service_rate is smaller than cold_service_rate

**analyze_custom_states**(*hist_states*, *skip_init_time=None*, *skip_init_index=None*)
Analyses a custom states list and calculates the amount of time spent in each state each time we enterred that state, and the times at which transitions have happened.

**Parameters**

- **hist_states** (*list[object]*) – The states calculated, should have the same dimensions as the *hist_** arrays.
- **skip_init_time** (*float, optional*) – The amount of time skipped in the beginning, by default None
- **skip_init_index** (*int, optional*) – The number of indices skipped in the beginning, by default None

**Returns** (residence_times, transition_times) where residence_times is an array of the amount of times we spent in each state, and transition_times are the moments of time at which each transition has occured

**Return type** list[float], list[float]

**calculate_time_average**(*values*, *skip_init_time=None*, *skip_init_index=None*)
calculate_time_average calculates the time-averaged of the values passed in with optional skipping a specific number of time steps (skip_init_index) and a specific amount of time (skip_init_time).

**Parameters**

- **values** (*list*) – A list of values with the same dimensions as history array (number of transitions)
- **skip_init_time** (*Float, optional*) – Amount of time skipped in the beginning to let the transient part of the solution pass, by default None
- **skip_init_index** (*[type], optional*) – Number of steps skipped in the beginning to let the transient behaviour of system pass, by default None

**Returns** returns (unq_vals, val_times) where unq_vals is the unique values inside the values list and val_times is the portion of the time that is spent in that value.

**Return type** (list, list)

**calculate_time_lengths**()
Calculate the time length for each step between two event transitions. Records the values in *self.time_lengths*.

**cold_start_arrival**(*t*)
Goes through the process necessary for a cold start arrival which includes generation of a new function instance in the *COLD* state and adding it to the cluster.

**Parameters** **t** (*float*) – The time at which the arrival has happened. This is used to record the creation time for the server and schedule the expiration of the instance if necessary.

**generate_trace**(*debug_print=False*, *progress=False*)
Generate a sample trace.

**Parameters**

- **debug_print** (*bool, optional*) – If True, will print each transition occuring during the simulation, by default False
- **progress** (*bool, optional*) – Whether or not the progress should be outputted using the *tqdm* library, by default False

**Raises** **Exception** – Raises if FunctionInstance enters an unknown state (other than *IDLE* for idle or *TERM* for terminated) after making an internal transition

**get_average_lifespan**()
Get the average lifespan of each instance, calculated by the amount of time from creation of instance, until its expiration.

**Returns** The average lifespan

**Return type** float

**get_average_residence_times**(*hist_states*, *skip_init_time=None*, *skip_init_index=None*)
Get the average residence time for each state in custom state encoding.

**Parameters**

- **hist_states** (*list[object]*) – The states calculated, should have the same dimensions as the *hist_** arrays.
- **skip_init_time** (*float, optional*) – The amount of time skipped in the beginning, by default None
- **skip_init_index** (*int, optional*) – The number of indices skipped in the beginning, by default None

**Returns** The average residence time for each state, averaged over the times we transitioned into that state

**Return type** float

**get_average_server_count**()
Get the time-average server count.

**Returns** Average server count

**Return type** float

**get_average_server_idle_count**()
Get the time-averaged idle server count.

**Returns** Average idle server count

**Return type** float

**get_average_server_running_count**()
Get the time-averaged running server count.

**Returns** Average running server coutn

**Return type** float

**get_cold_start_prob**()
Get the probability of cold start for the simulated trace.

**Returns** The probability of cold start calculated by dividing the number of cold start requests, over all requests

**Return type** float

**get_index_after_time**(*t*)
Get the first historical array index (for all arrays storing hisotrical events) that is after the time t.

**Parameters** **t** (*float*) – The time in the beginning we want to skip

**Returns** The calculated index in *self.hist_times*

**Return type** int

**get_request_custom_states**(*hist_states*, *skip_init_time=None*, *skip_init_index=None*)
Get request statistics for an array of custom states.

**Parameters**

- **hist_states** (*list[object]*) – An array of custom states calculated by the user for which the statistics should be calculated, should be the same size as *hist_** objects, these values will be used as the keys for the returned dataframe.
- **skip_init_time** (*float, optional*) – The amount of time skipped in the beginning, by default None
- **skip_init_index** (*int, optional*) – The number of indices that should be skipped in the beginning to calculate steady-state results, by default None

**Returns** A pandas dataframe including different statistics like *p_cold* (probability of cold start)

**Return type** pandas.DataFrame

**get_result_dict**()
Get the results of the simulation as a dict, which can easily be integrated into web services.

**Returns** A dictionary of different characteristics.

**Return type** dict

**get_skip_init**(*skip_init_time=None*, *skip_init_index=None*)
Get the minimum index which satisfies both the time and index count we want to skip in the beginning of the simulation, which is used to reduce the transient effect for calculating the steady-state values.

**Parameters**

- **skip_init_time** (*float, optional*) – The amount of time skipped in the beginning, by default None
- **skip_init_index** (*[type], optional*) – The number of indices we want to skip in the historical events, by default None

**Returns** The number of indices after which both index and time requirements are satisfied

**Return type** int

**get_trace_end**()
Get the time at which the trace (one iteration of the simulation) has ended. This mainly due to the fact that we keep on simulating until the trace time goes beyond max_time, but the time is incremented until the next event.

**Returns** The time at which the trace has ended

**Return type** float

**has_server**()
Returns True if there are still instances (servers) in the simulated platform, False otherwise.

**Returns** Whether or not the platform has instances (servers)

**Return type** bool

**is_warm_available**(*t*)
Whether we have at least one available instance in the warm pool that can process requests

**Parameters** **t** (*float*) – Current time

**Returns** True if at least one server is able to accept a request

**Return type** bool

**static print_time_average**(*vals*, *probs*, *column_width=15*)
Print the time average of states.

**Parameters**

- **vals** (*list[object]*) – The values for which the time average is to be printed
- **probs** (*list[float]*) – The probability of each of the members of the values array
- **column_width** (*int, optional*) – The width of the printed result for *vals*, by default 15

**print_trace_results**()
Print a brief summary of the results of the trace.

**req**()
Generate a request inter-arrival from *self.arrival_process*

**Returns** The generated inter-arrival sample

**Return type** float

**reset_trace**()
resets all the historical data to prepare the class for a new simulation

**schedule_warm_instance**(*t*)
Goes through a process to determine which warm instance should process the incoming request.

**Parameters** **t** (*float*) – The time at which the scheduling is happening

**Returns** The function instances that the scheduler has selected for the incoming request.

**Return type** *simfaas.FunctionInstance.FunctionInstance*

**trace_condition**(*t*)
The condition for resulting the trace, we continue the simulation until this function returns false.

**Parameters** **t** (*float*) – current time in the simulation since the start of simulation

**Returns** True if we should continue the simulation, false otherwise

**Return type** bool

**update_hist_arrays**(*t*)
Update history arrays

**Parameters** **t** (*float*) – Current time

**warm_start_arrival**(*t*)
Goes through the process necessary for a warm start arrival which includes selecting a warm instance for processing and recording the request information.

**Parameters** **t** (*float*) – The time at which the arrival has happened. This is used to record the creation time for the server and schedule the expiration of the instance if necessary.

CHAPTER

THREE

# TEMPORAL SIMULATOR

In this family of classes, we want to extract temporal characteristics using execution of simulations. We can extract average estimates by average over several executions of the simulation (sample average). All of these classes extend the functionality provided by *ServerlessSimulator*, you can use the same arguments and call the same methods, with some exteded functionality provided below.

## 3.1 Serverless Temporal Simulator

**class** simfaas.ServerlessTemporalSimulator.**ServerlessTemporalSimulator**(*running_function_instances*, *idle_function_instances*, **args*, ***kwargs*)

Bases: *simfaas.ServerlessSimulator.ServerlessSimulator*

ServerlessTemporalSimulator extends ServerlessSimulator to enable extraction of temporal characteristics. Also gets all of the arguments accepted by *ServerlessSimulator*

**Parameters**

- **running_function_instances** (*list[FunctionInstance]*) – A list containing the running function instances
- **idle_function_instances** (*list[FunctionInstance]*) – A list containing the idle function instances

## 3.2 Exponential Temporal Simulator

The exponential temporal simulator assume exponential inter-event distribution for both arrival and departure from each function instance.

**class** simfaas.ServerlessTemporalSimulator.**ExponentialServerlessTemporalSimulator**(*running_funct*
*idle_function_*
**args*,
***kwargs*)

Bases: *simfaas.ServerlessTemporalSimulator.ServerlessTemporalSimulator*

ExponentialServerlessTemporalSimulator is a simulator assuming exponential distribution for proceesing times which means each process is state-less and we can generate a service time and use that from now on. This class extends ServerlessTemporalSimulator which has functionality for other processes as well.

**Parameters**

- **running_function_instance_count** (*integer*) – running_function_instance_count is the number of instances currently processing a request
- **idle_function_instance_next_terminations** (*list[float]*) – idle_function_instance_next_terminations is an array of next termination scheduled for idle functions if they receive no new requests.

CHAPTER

# FOUR

# MULTI-REQUEST SERVERLESS SIMULATOR

This class represents simulators for serverless computing platforms like Google Cloud Run that allow multiple requests to enter an instance concurrenctly, and scale the instances based on a preset maximum concurrency value set by the user.

**class** simfaas.ParServerlessSimulator.**ParServerlessSimulator**(*concurrency_value: int*, **args*, ***kwargs*)

Bases: *simfaas.ServerlessSimulator.ServerlessSimulator*

ParServerlessSimulator is responsible for executing simulations of a sample serverless computing platform with the ability to handle concurrent request in each instance, mainly for the performance analysis and performance model evaluation purposes. For parameters, refer to *ServerlessSimulator*.

**Parameters** **concurrency_value** (*int*) – The number of concurrent requests allowed for each function instance.

**cold_start_arrival**(*t*)

Goes through the process necessary for a cold start arrival which includes generation of a new function instance in the *COLD* state and adding it to the cluster.

**Parameters** **t** (*float*) – The time at which the arrival has happened. This is used to record the creation time for the server and schedule the expiration of the instance if necessary.

**get_average_conc_avgs**()

Get the time-averaged average concurrency levels among all instances.

**Returns** Average concurrency levels of instances

**Return type** float

**get_result_dict**()

Get the results of the simulation as a dict, which can easily be integrated into web services.

**Returns** A dictionary of different characteristics.

**Return type** dict

**is_warm_available**(*t*)

Whether we have at least one available instance in the warm pool that can process requests

**Parameters** **t** (*float*) – Current time

**Returns** True if at least one server is able to accept a request

**Return type** bool

**print_trace_results**()

Print a brief summary of the results of the trace.

**reset_trace**()
resets all the historical data to prepare the class for a new simulation with additional functionality added to base class.

**update_hist_arrays**(*t*)
Update history arrays

**Parameters** **t** (*float*) – Current time

CHAPTER

FIVE

# SIMULATED PROCESS

**class** simfaas.SimProcess.**ConstSimProcess**(*rate*)
Bases: *simfaas.SimProcess.SimProcess*

ConstSimProcess extends the functionality of *SimProcess* for constant processes, meaning this is a deterministic process and fires exactly every *1/rate* seconds. This class does not implement the *pdf* and *cdf* functions.

**rate** [float] The rate at which the process should fire off

**generate_trace**()
generate_trace function is supposed to be replaced with the override function of each of the child classes.

**NotImplementedError** By default, this function raises NotImplementedError unless overriden by a child class.

**class** simfaas.SimProcess.**ExpSimProcess**(*rate*)
Bases: *simfaas.SimProcess.SimProcess*

ExpSimProcess extends the functionality of *SimProcess* for exponentially distributed processes. This class also implements the *pdf* and *cdf* functions which can be used for visualization purposes.

**rate** [float] The rate at which the process should fire off

**cdf**(*x*)
cdf function is called for visualization for classes with *self.has_cdf = True*.

**Parameters** **x** (*float*) – The time for which the cdf value (density) should be returned

**Raises** **NotImplementedError** – By default, this function raises NotImplementedError unless overriden by a child class.

**generate_trace**()
generate_trace function is supposed to be replaced with the override function of each of the child classes.

**NotImplementedError** By default, this function raises NotImplementedError unless overriden by a child class.

**pdf**(*x*)
pdf function is called for visualization for classes with *self.has_pdf = True*.

**Parameters** **x** (*float*) – The time for which the pdf value (density) should be returned

**Raises** **NotImplementedError** – By default, this function raises NotImplementedError unless overriden by a child class.

**class** simfaas.SimProcess.**GaussianSimProcess**(*rate*, *std*)
Bases: *simfaas.SimProcess.SimProcess*

GaussianSimProcess extends the functionality of *SimProcess* for gaussian processes. This class also implements the *pdf* and *cdf* functions which can be used for visualization purposes.

**rate** [float] The rate at which the process should fire off

**std** [float] The standard deviation of the simulated process

**cdf**(*x*)
cdf function is called for visualization for classes with *self.has_cdf = True*.

**Parameters** **x** (*float*) – The time for which the cdf value (density) should be returned

**Raises** **NotImplementedError** – By default, this function raises NotImplementedError unless overriden by a child class.

**generate_trace**()
generate_trace function is supposed to be replaced with the override function of each of the child classes.

**NotImplementedError** By default, this function raises NotImplementedError unless overriden by a child class.

**pdf**(*x*)
pdf function is called for visualization for classes with *self.has_pdf = True*.

**Parameters** **x** (*float*) – The time for which the pdf value (density) should be returned

**Raises** **NotImplementedError** – By default, this function raises NotImplementedError unless overriden by a child class.

**class** simfaas.SimProcess.**SimProcess**
Bases: object

SimProcess gives us a single interface to simulate different processes. This will later on be used to simulated different processes and compare them agaist a custom analytical model. In the child class, after performing *super().__init__()*, properties *self.has_pdf* and *self.has_cdf* by default value of *False* will be created. In case your class has the proposed PDF and CDF functions available, you need to override these values in order for your model PDF to show up in the output plot.

**cdf**(*x*)
cdf function is called for visualization for classes with *self.has_cdf = True*.

**Parameters** **x** (*float*) – The time for which the cdf value (density) should be returned

**Raises** **NotImplementedError** – By default, this function raises NotImplementedError unless overriden by a child class.

**generate_trace**()
generate_trace function is supposed to be replaced with the override function of each of the child classes.

**NotImplementedError** By default, this function raises NotImplementedError unless overriden by a child class.

**pdf**(*x*)
pdf function is called for visualization for classes with *self.has_pdf = True*.

**Parameters** **x** (*float*) – The time for which the pdf value (density) should be returned

**Raises** **NotImplementedError** – By default, this function raises NotImplementedError unless overriden by a child class.

**visualize**(*num_traces=10000*, *num_bins=100*)
visualize function visualizes the PDF and CDF of the simulated process by generating traces from your function using *generate_trace()* and converting the resulting histogram values (event counts) to densities to be comparable with PDF and CDF functions calculated analytically.

**num_traces** [int, optional] Number of traces we want to generate for calculating the histogram, by default 10000

**num_bins** [int, optional] Number of bins for the histogram which created the density probabilities, by default 100

CHAPTER

SIX

# API REFERENCE

You can also checkout Module Index for a list of all modules implemented in this package.

## 6.1 ServerlessSimulator

**class** simfaas.ServerlessSimulator.**ServerlessSimulator**(*arrival_process=None*, *warm_service_process=None*, *cold_service_process=None*, *expiration_threshold=600*, *max_time=86400*, *maximum_concurrency=1000*, ***kwargs*)

Bases: object

ServerlessSimulator is responsible for executing simulations of a sample serverless computing platform, mainly for the performance analysis and performance model evaluation purposes.

**Parameters**

- **arrival_process** (simfaas.SimProcess.SimProcess, *optional*) – The process used for generating inter-arrival samples, if absent, *arrival_rate* should be passed to signal exponential distribution, by default None
- **warm_service_process** (simfaas.SimProcess.SimProcess, *optional*) – The process which will be used to calculate service times, if absent, *warm_service_rate* should be passed to signal exponential distribution, by default None
- **cold_service_process** (simfaas.SimProcess.SimProcess, *optional*) – The process which will be used to calculate service times, if absent, *cold_service_rate* should be passed to signal exponential distribution, by default None
- **expiration_threshold** (*float, optional*) – The period of time after which the instance will be expired and the capacity release for use by others, by default 600
- **max_time** (*float, optional*) – The maximum amount of time for which the simulation should continue, by default 24*60*60 (24 hours)
- **maximum_concurrency** (*int, optional*) – The maximum number of concurrently executing function instances allowed on the system This will be used to determine when a rejection of request should happen due to lack of capacity, by default 1000

**Raises**

- **Exception** – Raises if neither arrival_process nor arrival_rate are present
- **Exception** – Raises if neither warm_service_process nor warm_service_rate are present

- **Exception** – Raises if neither cold_service_process nor cold_service_rate are present
- **ValueError** – Raises if warm_service_rate is smaller than cold_service_rate

**analyze_custom_states**(*hist_states*, *skip_init_time=None*, *skip_init_index=None*)

Analyses a custom states list and calculates the amount of time spent in each state each time we enterred that state, and the times at which transitions have happened.

**Parameters**

- **hist_states** (*list[object]*) – The states calculated, should have the same dimensions as the *hist_** arrays.
- **skip_init_time** (*float, optional*) – The amount of time skipped in the beginning, by default None
- **skip_init_index** (*int, optional*) – The number of indices skipped in the beginning, by default None

**Returns** (residence_times, transition_times) where residence_times is an array of the amount of times we spent in each state, and transition_times are the moments of time at which each transition has occured

**Return type** list[float], list[float]

**calculate_time_average**(*values*, *skip_init_time=None*, *skip_init_index=None*)

calculate_time_average calculates the time-averaged of the values passed in with optional skipping a specific number of time steps (skip_init_index) and a specific amount of time (skip_init_time).

**Parameters**

- **values** (*list*) – A list of values with the same dimensions as history array (number of transitions)
- **skip_init_time** (*Float, optional*) – Amount of time skipped in the beginning to let the transient part of the solution pass, by default None
- **skip_init_index** (*[type], optional*) – Number of steps skipped in the beginning to let the transient behaviour of system pass, by default None

**Returns** returns (unq_vals, val_times) where unq_vals is the unique values inside the values list and val_times is the portion of the time that is spent in that value.

**Return type** (list, list)

**calculate_time_lengths**()

Calculate the time length for each step between two event transitions. Records the values in *self.time_lengths*.

**cold_start_arrival**(*t*)

Goes through the process necessary for a cold start arrival which includes generation of a new function instance in the *COLD* state and adding it to the cluster.

**Parameters** **t** (*float*) – The time at which the arrival has happened. This is used to record the creation time for the server and schedule the expiration of the instance if necessary.

**generate_trace**(*debug_print=False*, *progress=False*)

Generate a sample trace.

**Parameters**

- **debug_print** (*bool, optional*) – If True, will print each transition occuring during the simulation, by default False

- **progress** (*bool, optional*) – Whether or not the progress should be outputted using the *tqdm* library, by default False

**Raises** **Exception** – Raises if FunctionInstance enters an unknown state (other than *IDLE* for idle or *TERM* for terminated) after making an internal transition

**get_average_lifespan**()
Get the average lifespan of each instance, calculated by the amount of time from creation of instance, until its expiration.

**Returns** The average lifespan

**Return type** float

**get_average_residence_times**(*hist_states*, *skip_init_time=None*, *skip_init_index=None*)
Get the average residence time for each state in custom state encoding.

**Parameters**

- **hist_states** (*list[object]*) – The states calculated, should have the same dimensions as the *hist_** arrays.
- **skip_init_time** (*float, optional*) – The amount of time skipped in the beginning, by default None
- **skip_init_index** (*int, optional*) – The number of indices skipped in the beginning, by default None

**Returns** The average residence time for each state, averaged over the times we transitioned into that state

**Return type** float

**get_average_server_count**()
Get the time-average server count.

**Returns** Average server count

**Return type** float

**get_average_server_idle_count**()
Get the time-averaged idle server count.

**Returns** Average idle server count

**Return type** float

**get_average_server_running_count**()
Get the time-averaged running server count.

**Returns** Average running server coutn

**Return type** float

**get_cold_start_prob**()
Get the probability of cold start for the simulated trace.

**Returns** The probability of cold start calculated by dividing the number of cold start requests, over all requests

**Return type** float

**get_index_after_time**(*t*)
Get the first historical array index (for all arrays storing hisotrical events) that is after the time t.

**Parameters** **t** (*float*) – The time in the beginning we want to skip

**Returns** The calculated index in *self.hist_times*

**Return type** int

**get_request_custom_states**(*hist_states*, *skip_init_time=None*, *skip_init_index=None*)
Get request statistics for an array of custom states.

**Parameters**

- **hist_states** (*list[object]*) – An array of custom states calculated by the user for which the statistics should be calculated, should be the same size as *hist_** objects, these values will be used as the keys for the returned dataframe.
- **skip_init_time** (*float, optional*) – The amount of time skipped in the beginning, by default None
- **skip_init_index** (*int, optional*) – The number of indices that should be skipped in the beginning to calculate steady-state results, by default None

**Returns** A pandas dataframe including different statistics like *p_cold* (probability of cold start)

**Return type** pandas.DataFrame

**get_result_dict**()
Get the results of the simulation as a dict, which can easily be integrated into web services.

**Returns** A dictionary of different characteristics.

**Return type** dict

**get_skip_init**(*skip_init_time=None*, *skip_init_index=None*)
Get the minimum index which satisfies both the time and index count we want to skip in the beginning of the simulation, which is used to reduce the transient effect for calculating the steady-state values.

**Parameters**

- **skip_init_time** (*float, optional*) – The amount of time skipped in the beginning, by default None
- **skip_init_index** (*[type], optional*) – The number of indices we want to skip in the historical events, by default None

**Returns** The number of indices after which both index and time requirements are satisfied

**Return type** int

**get_trace_end**()
Get the time at which the trace (one iteration of the simulation) has ended. This mainly due to the fact that we keep on simulating until the trace time goes beyond max_time, but the time is incremented until the next event.

**Returns** The time at which the trace has ended

**Return type** float

**has_server**()
Returns True if there are still instances (servers) in the simulated platform, False otherwise.

**Returns** Whether or not the platform has instances (servers)

**Return type** bool

**is_warm_available**(*t*)
Whether we have at least one available instance in the warm pool that can process requests

**Parameters** **t** (*float*) – Current time

**Returns** True if at least one server is able to accept a request

**Return type** bool

**static print_time_average**(*vals*, *probs*, *column_width=15*)
Print the time average of states.

**Parameters**

- **vals** (*list[object]*) – The values for which the time average is to be printed
- **probs** (*list[float]*) – The probability of each of the members of the values array
- **column_width** (*int, optional*) – The width of the printed result for *vals*, by default 15

**print_trace_results**()
Print a brief summary of the results of the trace.

**req**()
Generate a request inter-arrival from *self.arrival_process*

**Returns** The generated inter-arrival sample

**Return type** float

**reset_trace**()
resets all the historical data to prepare the class for a new simulation

**schedule_warm_instance**(*t*)
Goes through a process to determine which warm instance should process the incoming request.

**Parameters** **t** (*float*) – The time at which the scheduling is happening

**Returns** The function instances that the scheduler has selected for the incoming request.

**Return type** *simfaas.FunctionInstance.FunctionInstance*

**trace_condition**(*t*)
The condition for resulting the trace, we continue the simulation until this function returns false.

**Parameters** **t** (*float*) – current time in the simulation since the start of simulation

**Returns** True if we should continue the simulation, false otherwise

**Return type** bool

**update_hist_arrays**(*t*)
Update history arrays

**Parameters** **t** (*float*) – Current time

**warm_start_arrival**(*t*)
Goes through the process necessary for a warm start arrival which includes selecting a warm instance for processing and recording the request information.

**Parameters** **t** (*float*) – The time at which the arrival has happened. This is used to record the creation time for the server and schedule the expiration of the instance if necessary.

## 6.2 ServerlessTemporalSimulator

**class** simfaas.ServerlessTemporalSimulator.**ExponentialServerlessTemporalSimulator**(*running_funct*
*idle_function_*
**args*,
***kwargs*)

Bases: *simfaas.ServerlessTemporalSimulator.ServerlessTemporalSimulator*

ExponentialServerlessTemporalSimulator is a simulator assuming exponential distribution for proceesing times which means each process is state-less and we can generate a service time and use that from now on. This class extends ServerlessTemporalSimulator which has functionality for other processes as well.

**Parameters**

- **running_function_instance_count** (*integer*) – running_function_instance_count is the number of instances currently processing a request
- **idle_function_instance_next_terminations** (*list[float]*) – idle_function_instance_next_terminations is an array of next termination scheduled for idle functions if they receive no new requests.

**class** simfaas.ServerlessTemporalSimulator.**ServerlessTemporalSimulator**(*running_function_instances*,
*idle_function_instances*,
**args*,
***kwargs*)

Bases: *simfaas.ServerlessSimulator.ServerlessSimulator*

ServerlessTemporalSimulator extends ServerlessSimulator to enable extraction of temporal characteristics. Also gets all of the arguments accepted by *ServerlessSimulator*

**Parameters**

- **running_function_instances** (*list[FunctionInstance]*) – A list containing the running function instances
- **idle_function_instances** (*list[FunctionInstance]*) – A list containing the idle function instances

## 6.3 ParServerlessSimulator

**class** simfaas.ParServerlessSimulator.**ParServerlessSimulator**(*concurrency_value: int*, **args*,
***kwargs*)

Bases: *simfaas.ServerlessSimulator.ServerlessSimulator*

ParServerlessSimulator is responsible for executing simulations of a sample serverless computing platform with the ability to handle concurrent request in each instance, mainly for the performance analysis and performance model evaluation purposes. For parameters, refer to *ServerlessSimulator*.

**Parameters** **concurrency_value** (*int*) – The number of concurrent requests allowed for each function instance.

**cold_start_arrival**(*t*)

Goes through the process necessary for a cold start arrival which includes generation of a new function instance in the *COLD* state and adding it to the cluster.

**Parameters** **t** (*float*) – The time at which the arrival has happened. This is used to record the creation time for the server and schedule the expiration of the instance if necessary.

**get_average_conc_avgs**()
Get the time-averaged average concurrency levels among all instances.

**Returns** Average concurrency levels of instances

**Return type** float

**get_result_dict**()
Get the results of the simulation as a dict, which can easily be integrated into web services.

**Returns** A dictionary of different characteristics.

**Return type** dict

**is_warm_available**(*t*)
Whether we have at least one available instance in the warm pool that can process requests

**Parameters** **t** (*float*) – Current time

**Returns** True if at least one server is able to accept a request

**Return type** bool

**print_trace_results**()
Print a brief summary of the results of the trace.

**reset_trace**()
resets all the historical data to prepare the class for a new simulation with additional functionality added to base class.

**update_hist_arrays**(*t*)
Update history arrays

**Parameters** **t** (*float*) – Current time

## 6.4 FunctionInstance

**class** simfaas.FunctionInstance.**FunctionInstance**(*t*, *cold_service_process*, *warm_service_process*, *expiration_threshold*)

Bases: object

FunctionInstance aims to simulate the behaviour of a function instance in a serverless platform, with all the internal transitions necessary.

**Parameters**

- **t** (*float*) – The time at which the instance is being created
- **cold_service_process** (simfaas.SimProcess.SimProcess) – The process used to sample cold start response times
- **warm_service_process** (simfaas.SimProcess.SimProcess) – The process used to sample warm start response times
- **expiration_threshold** (*float*) – The amount of time it takes for an instance to get expired and the resources consumed by it released after processing the last request

**arrival_transition**(*t*)
Make an arrival transition, which causes the instance to go from IDLE to WARM

**Parameters** **t** (*float*) – The time at which the transition has occured, this also updates the next termination.

**Raises** **Exception** – Raises if currently process a request by being in *COLD* or *WARM* states

**generate_cold_departure**(*t*)
generate the departure of the cold request which is the first request received by the instance.

**Parameters** **t** (*float*) – Current time in simulation

**get_life_span**()
Get the life span of the server, e.g. after the server has been terminated

**Returns** life span of the instance

**Return type** float

**get_next_departure**(*t*)
Get the time until the next departure

**Parameters** **t** (*float*) – Current time

**Returns** Amount of time until the next departure

**Return type** float

**Raises** **Exception** – Raises if called after the departure

**get_next_termination**(*t*)
Get the time until the next termination

**Parameters** **t** (*float*) – Current time

**Returns** Amount of time until the next termination

**Return type** float

**Raises** **Exception** – Raises if called after the termination

**get_next_transition_time**(*t=0*)
Get how long until the next transition.

**Parameters** **t** (*float, optional*) – The current time, by default 0

**Returns** The seconds remaining until the next transition

**Return type** float

**get_state**()
Get the current state

**Returns** currentstate

**Return type** str

**is_idle**()
Whether or not the instance is currently idle, and thus can accept new requests.

**Returns** True if idle, false otherwise

**Return type** bool

**is_ready**()
Whether or not the instance is ready to accept new requests. Here, same as is_idle()

**Returns** True if ready to accept new requests, False otherwise

**Return type** bool

**make_transition**()
Make the next internal transition, either transition into *IDLE* of already processing a request, or *TERM* if scheduled termination has arrived.

**Returns** The state after making the internal transition

**Return type** str

**Raises** **Exception** – Raises if already in *TERM* state, since no other internal transitions are possible

**update_next_termination**()
Update the next scheduled termination if no other requests are made to the instance.

## 6.5 ParFunctionInstance

**class** simfaas.ParFunctionInstance.**ParFunctionInstance**(*concurrency_value*, **args*, ***kwargs*)

Bases: *simfaas.FunctionInstance.FunctionInstance*

ParFunctionInstance aims to simulate the behaviour of a function instance in a serverless platform, with all the internal transitions necessary allowing multiple requests to be parsed. For other input parameters, refer to *FunctionInstance*.

**Parameters**

- **cold_service_process** (simfaas.SimProcess.SimProcess) – The process used to sample cold start *initialization* process before warm process is starting. Note that this is different from *FunctionInstance*
- **concurrency_value** (*int*) – The number of parallel requests that a single instance can handle.

**arrival_transition**(*t*)
Make an arrival transition, which causes the instance to go from IDLE to WARM

**Parameters** **t** (*float*) – The time at which the transition has occured, this also updates the next termination.

**Raises** **Exception** – Raises if currently process a request by being in *COLD* or *WARM* states

**generate_cold_departure**(*t*)
generate the departure of the cold request which is the first request received by the instance.

**Parameters** **t** (*float*) – Current time in simulation

**get_concurrency**()
get current concurrency level

**Returns** number of requests being processed right now

**Return type** int

**get_next_departure**(*t*)
Get the time until the next departure

**Parameters** **t** (*float*) – Current time

**Returns** Amount of time until the next departure

**Return type** float

**Raises** **Exception** – Raises if called after the departure

**get_next_transition_time**(*t=0*)
Get how long until the next transition.

**Parameters** **t** (*float, optional*) – The current time, by default 0

**Returns** The seconds remaining until the next transition

**Return type** float

**is_ready**()
Whether or not the instance is ready to accept new requests. Here, same as is_idle()

**Returns** True if ready to accept new requests, False otherwise

**Return type** bool

**make_transition**()
Make the next internal transition, either transition into *IDLE* of already processing a request, or *TERM* if scheduled termination has arrived.

**Returns** The state after making the internal transition

**Return type** str

**Raises** **Exception** – Raises if already in *TERM* state, since no other internal transitions are possible

**update_next_termination**()
Update the next scheduled termination if no other requests are made to the instance.

## 6.6 SimProcess

**class** simfaas.SimProcess.**ConstSimProcess**(*rate*)
Bases: *simfaas.SimProcess.SimProcess*

ConstSimProcess extends the functionality of *SimProcess* for constant processes, meaning this is a deterministic process and fires exactly every *1/rate* seconds. This class does not implement the *pdf* and *cdf* functions.

**rate** [float] The rate at which the process should fire off

**generate_trace**()
generate_trace function is supposed to be replaced with the override function of each of the child classes.

**NotImplementedError** By default, this function raises NotImplementedError unless overriden by a child class.

**class** simfaas.SimProcess.**ExpSimProcess**(*rate*)
Bases: *simfaas.SimProcess.SimProcess*

ExpSimProcess extends the functionality of *SimProcess* for exponentially distributed processes. This class also implements the *pdf* and *cdf* functions which can be used for visualization purposes.

**rate** [float] The rate at which the process should fire off

**cdf**(*x*)
cdf function is called for visualization for classes with *self.has_cdf = True*.

**Parameters** **x** (*float*) – The time for which the cdf value (density) should be returned

**Raises** **NotImplementedError** – By default, this function raises NotImplementedError unless overriden by a child class.

**generate_trace**()
generate_trace function is supposed to be replaced with the override function of each of the child classes.

**NotImplementedError** By default, this function raises NotImplementedError unless overriden by a child class.

**pdf**(*x*)
pdf function is called for visualization for classes with *self.has_pdf = True*.

**Parameters** **x** (*float*) – The time for which the pdf value (density) should be returned

**Raises** **NotImplementedError** – By default, this function raises NotImplementedError unless overriden by a child class.

**class** simfaas.SimProcess.**GaussianSimProcess**(*rate*, *std*)
Bases: *simfaas.SimProcess.SimProcess*

GaussianSimProcess extends the functionality of *SimProcess* for gaussian processes. This class also implements the *pdf* and *cdf* functions which can be used for visualization purposes.

**rate** [float] The rate at which the process should fire off

**std** [float] The standard deviation of the simulated process

**cdf**(*x*)
cdf function is called for visualization for classes with *self.has_cdf = True*.

**Parameters** **x** (*float*) – The time for which the cdf value (density) should be returned

**Raises** **NotImplementedError** – By default, this function raises NotImplementedError unless overriden by a child class.

**generate_trace**()
generate_trace function is supposed to be replaced with the override function of each of the child classes.

**NotImplementedError** By default, this function raises NotImplementedError unless overriden by a child class.

**pdf**(*x*)
pdf function is called for visualization for classes with *self.has_pdf = True*.

**Parameters** **x** (*float*) – The time for which the pdf value (density) should be returned

**Raises** **NotImplementedError** – By default, this function raises NotImplementedError unless overriden by a child class.

**class** simfaas.SimProcess.**SimProcess**
Bases: object

SimProcess gives us a single interface to simulate different processes. This will later on be used to simulated different processes and compare them agaist a custom analytical model. In the child class, after performing *super().__init__()*, properties *self.has_pdf* and *self.has_cdf* by default value of *False* will be created. In case your class has the proposed PDF and CDF functions available, you need to override these values in order for your model PDF to show up in the output plot.

**cdf**(*x*)
cdf function is called for visualization for classes with *self.has_cdf = True*.

**Parameters** **x** (*float*) – The time for which the cdf value (density) should be returned

**Raises** **NotImplementedError** – By default, this function raises NotImplementedError unless overriden by a child class.

**generate_trace**()
generate_trace function is supposed to be replaced with the override function of each of the child classes.

**NotImplementedError** By default, this function raises NotImplementedError unless overriden by a child class.

**pdf**(*x*)
pdf function is called for visualization for classes with *self.has_pdf = True*.

**Parameters** **x** (*float*) – The time for which the pdf value (density) should be returned

**Raises** **NotImplementedError** – By default, this function raises NotImplementedError unless overriden by a child class.

**visualize**(*num_traces=10000*, *num_bins=100*)
visualize function visualizes the PDF and CDF of the simulated process by generating traces from your function using *generate_trace()* and converting the resulting histogram values (event counts) to densities to be comparable with PDF and CDF functions calculated analytically.

**num_traces** [int, optional] Number of traces we want to generate for calculating the histogram, by default 10000

**num_bins** [int, optional] Number of bins for the histogram which created the density probabilities, by default 100

## 6.7 Utility

simfaas.Utility.**convert_hist_pdf**(*_values*, *num_bins*)
convert_hist_pdf converts the histogram resulting from _values and num_bins to a density plot by dividing the probability of falling into a bin by the bin size, converting the values to density. The resulting values could be plotted and compared with the analytical pdf and cdf functions.

**_values** [list[float]] A list of values that we want to analyze and calculate the histogram for

**num_bins** [int] Number of bins used for generating the histogram

**list[float], list[float], list[float]** base, values, cumulative are returned which are the histogram bases, density values, and cumulative densities which can be compared with the analytical cdf function

CHAPTER

SEVEN

# JUPYTER DOCKER IMAGE

To ease the process of installation and experimenttion with `SimFaaS`, we developed a docker image extending the Jupyter Notebook Data Science Stack. The resulting docker image is also available publicly on Docker Hub.

## 7.1 Installation

The only requirement for running the jupyter notebook stack is `docker` which can easily be installed:

```
sudo apt-get update && sudo apt-get -y install docker.io

docker ps
sudo docker ps

sudo usermod -aG docker $USER
sudo chown "$USER":"$USER" /home/"$USER"/.docker -R
sudo chmod g+rwx "/home/$USER/.docker" -R
sudo chown "$USER":"$USER" /var/run/docker.sock
sudo chmod g+rwx /var/run/docker.sock -R
sudo systemctl enable docker
```

## 7.2 Running

To run the jupyter lab in the `current directory`, simply run the following command:

```
IMAGE_NAME=nimamahmoudi/jupyter-simfaas # or $(cat .dockername) if in root folder of␣
↪the github repo
TARGET_PORT=8888 # The port on which the jupyter notebook will run

docker run -it --rm \
    -p $TARGET_PORT:8888 \
    -e JUPYTER_ENABLE_LAB=yes \
    --name jpsimfaas \
    -v "$(pwd)":/home/jovyan/work \
    $IMAGE_NAME
```

The container logs will contain the token you need to log into your jupyter lab session.

## 7.3 Building the Docker Image

In case you made changes to the library and wanted to build the docker image with your own changes, you can run the following command.

```
IMAGE_NAME=nimamahmoudi/jupyter-simfaas # or $(cat .dockername) if in root folder of␣
↪the github repo
docker build . -t $IMAGE_NAME
```

CHAPTER

EIGHT

# INDICES AND TABLES

- genindex
- modindex
- search

# PYTHON MODULE INDEX

## S

# INDEX

## H



## I



## M



## P



## R

## S



## T



## U



## V



## W