Simulation and Modelling
Serverless Cold Starts

Zack Reeves (CID: 01844549)

November 2022

Contents

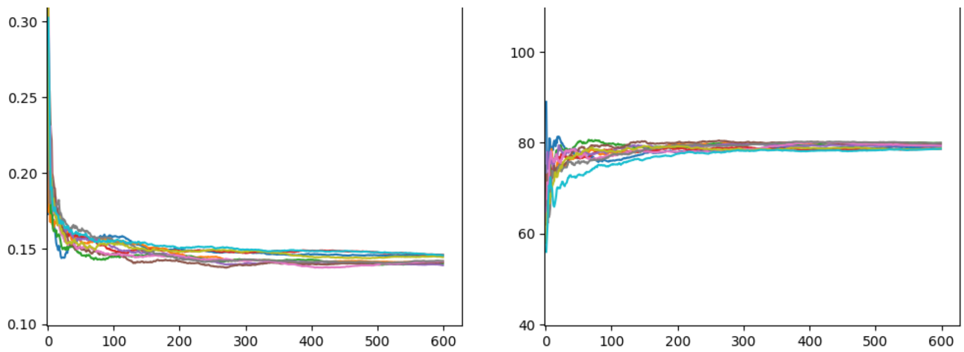
1 Simulation

1.1 Cold Start Ratio and Loss Rate

Python and a module, SimPy, were used to create a simulation of the FaaS system (see Appendix A for full code). The simulation was used to produce results for the following measures:

- C_{ratio}: No. cold starts/No. requests
- L_{rate}: No. lost requests/time

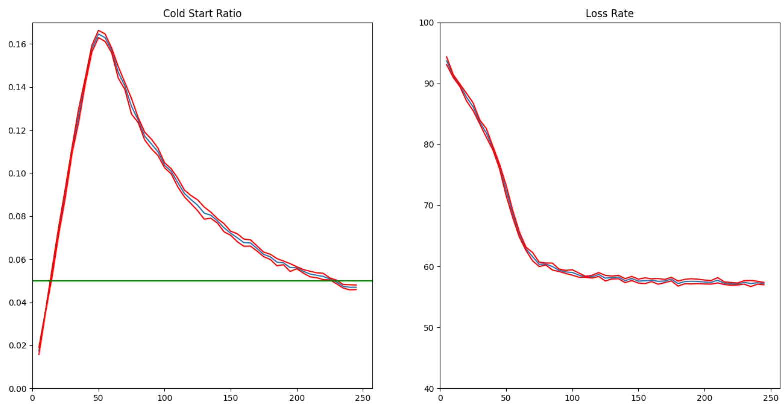
With capacity M = 40 and a simulation time of 10 minutes, the following data was produced



	Point Estimate	90% Confidence Intervals (10 individual replications)
Cold Start Ratio	0.1426	(0.1413, 0.1440)
Loss Rate	79.3631	(79.0700, 79.6562)

1.2 Changing the Capacity of M

The following data is produced by varying M from 5 to 245. It was produced using a simulation time of 5 minutes with 5 individual replications. Appendix B contains all the point estimates and confidence intervals.



At M = 230 the point estimate for the cold start ratio drops below 5% and at M = 235 the upper confidence interval drops below 5%.

1.3 Modelling Request Generation and the Passage of Time

As each function has an exponentially distributed inter-arrival time, the overall request generation can be managed by an aggregated Poisson process. When parsing the CSV trace, each function's invocations can be divided by 2592000 (conversion from 30 days to seconds). This gives the rate parameter for each function λ_f. Merging Poisson processes is done by summing the individual rate parameters.

$$\lambda = \sum_{f=1}^{10861} \lambda_f$$

Now the arrival processes have been merged, an exponential sample can be taken to determine the time until the next request. With a new request, a discrete distribution can be sampled to determine which function the request is for. The discrete distribution has buckets from 1 to 10861, one for each function, and the weights can be determined as

$$w_f = \frac{\lambda_f}{\sum_{f=1}^{10861} \lambda_f}$$

The output of this sample is the function that has made the request and the simulation can continue to determine whether the request is lost, incurs a cold start or starts being serviced. If a cold start is incurred an exponential sample is taken from the cold start distribution with $\lambda_{cold\ start} = 0.5$ to give the cold startup time before being serviced. Once the request is ready to be serviced, the function's service time is obtained by sampling an exponential distribution with

$$\lambda = \frac{1}{Avg.\ service\ time\ (in\ seconds)}$$

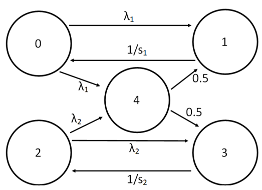
After this service time period, the request is complete. This is the process each request goes through from generation to loss/completion.

2 Analytical Modelling

2.1 Defining the State Space

State 0: Function 1 is idle in memory
State 1: Function 1 is executing in memory
State 2: Function 2 is idle in memory
State 3: Function 2 is executing in memory
State 4: Either function is in cold start

2.2 CTMC Transition Diagram



λ_i = arrival rate of requests for function i
s_i = avg. service time for function i
0.5 is the cold startup rate

Infinitesimal Generator Matrix

$$Q = \begin{pmatrix} -(\lambda_1+\lambda_2) & \lambda_1 & 0 & 0 & \lambda_2 \\ \frac{1}{s_1} & -(\frac{1}{s_1}) & 0 & 0 & 0 \\ 0 & 0 & -(\lambda_1+\lambda_2) & \lambda_2 & \lambda_1 \\ 0 & 0 & \frac{1}{s_2} & -(\frac{1}{s_2}) & 0 \\ 0 & 0.5 & 0 & 0.5 & -1 \end{pmatrix}$$

2.3 Defining Measures

$C_{ratio} = \frac{P_0 \cdot \lambda_2 + P_2 \cdot \lambda_1}{\lambda_1 + \lambda_2}$

The rate of cold starts for function 2 plus the rate of cold starts for function 1 over the rate of requests

$L_{rate} = (P_1 + P_3 + P_4) \cdot (\lambda_1 + \lambda_2)$

The probability the server is already executing or in cold start times the rate of requests

2.4 Solving the CTMC

Solving the above CTMC yields the following probabilities:

Inputs	Probabilities	Measures
$\lambda_1 = 0.4133858025$	$P_0 = 0.7234592085512606$	$C_{ratio} = 0.25001658045186903$
$\lambda_2 = 0.08635069444$	$P_1 = 4.700006616908895E\text{-}4$	$L_{rate} = 0.06267693284502052$
$s_1 = 0.0013$	$P_2 = 0.1511208286707441$	
$s_2 = 0.0001$	$P_3 = 7.552059352854275E\text{-}6$	
	$P_4 = 0.12494241005695148$	

A Full Code

A.1 FaaS Simulation

Beneath is the full code used to simulate the FaaS system using python and SimPy.

```
import csv
import simpy
import numpy as np
import datetime
import matplotlib.pyplot as plt


def parse_csv(file):
    # parses csv into to lists
    # convert service times to seconds
    # convert invocations into invocations/second

    with open(file, mode='r') as csv_file:
        csv_reader = csv.DictReader(csv_file)
        lines = 0
        lam = 0
        sum_lam = 0
        avg_service_times = []
        avg_arrival_rates = []
        weights = []
        for row in csv_reader:
            avg_service_times.append(int(row["AvgServiceTimeMillisec"])/1000)
            lam = int(row["Invocations30Days"])/(30*24*60*60)
            avg_arrival_rates.append(lam)
            sum_lam += lam
            lines += 1

        # compiles a list of weights to be used in a discrete distribution sampler
        for i in range(len(avg_service_times)):
            weights.append(avg_arrival_rates[i]/sum_lam)


    return avg_service_times, weights, sum_lam


class g:
    no_runs = 1 # used to repeat with different M
    no_trials = 5 # repeats each model this many times
    sim_duration = 60*60*24 # runs sim for this many simulated seconds
    f = 10861 # no. functions
    cold_start = 0.5 #c old start rate
    avg_service_times, weights, sum_lam = parse_csv('trace-final.csv')


class Request:
    def __init__(self, f_id):
        self.id = f_id


class Faas_Model:

    def __init__(self, trial_no, m, file_name):
        self.env = simpy.Environment()

        self.trial_no = trial_no
        self.m = m
        self.results_file = file_name

        # tracks status of each f 0: not in m, 1: idle, 2: executing
        self.status = np.ones(self.m).tolist() + np.zeros(g.f-self.m).tolist()

        # initializes memory with the first m functions all idle
        self.idle_queue = np.arange(self.m).tolist()


        # measures
        self.request_counter = 0
        self.cold_start_counter = 0
        self.lost_request_counter = 0
        self.memory_full_loss = 0
        self.already_running_loss = 0
```

```
        self.completions = 0

        # lists for graphs
        self.obs_time = []
        self.obs_cold_ratio = []
        self.obs_loss_rate = []

    def generate_requests(self):
        # generates requests using aggregate poisson process
        # sample a discrete distribution to assign the request a function

        while True:

            yield self.env.timeout(generate_time_to_request())

            self.request_counter += 1

            r_id = int(np.random.choice(a=g.f, p=g.weights)) # split poisson to stream f
            r = Request(r_id)

            self.env.process(self.service_request(r))

    def service_request(self, request):

        # check if f is running and reject request
        if self.status[request.id] == 2:
            self.already_running_loss += 1
            self.lost_request_counter += 1

        # check if f is idle and start executing
        elif self.status[request.id] == 1:

            self.status[request.id] = 2
            self.env.process(self.complete_request(request))

        # if f is not in memory check if everything in memory is executing
        else:
            self.mem_in_use = self.status.count(2)

            if self.mem_in_use >= self.m:
                # m is full reject request
                self.memory_full_loss += 1
                self.lost_request_counter += 1


            else:
                # Check top of the idle queue for which function to remove
                evict = self.idle_queue[0]
                self.status[evict] = 0
                self.idle_queue = remove_values_from_list(self.idle_queue, evict)

                self.status[request.id] = 2

                self.cold_start_counter += 1

                # sample cold start time
                yield self.env.timeout(generate_time_to_start())

                self.env.process(self.complete_request(request))


    def complete_request(self, request):
        # remove f from idle queue
        self.idle_queue = remove_values_from_list(self.idle_queue, request.id)

        # sample service time
        yield self.env.timeout(generate_time_to_service(request.id))

        self.completions += 1

        # return to idling, add f to the end of the idle queue
        self.status[request.id] = 1
        self.idle_queue.append(request.id)

    def observe(self):
        # records measure every second
```

```python
        while True:
            try:

                self.obs_cold_ratio.append(self.cold_start_counter/self.request_counter)
                self.obs_time.append(self.env.now)
                self.obs_loss_rate.append(self.lost_request_counter/self.obs_time[-1])
            except:
                pass
            yield self.env.timeout(1)



    def run(self):

        # start sim
        self.env.process(self.generate_requests())
        self.env.process(self.observe())
        self.env.run(until=g.sim_duration)

        # Output to terminal
        print(f'\nTrial {self.trial_no+1} of {g.no_trials}')
        print(f'Simulated for {str(datetime.timedelta(seconds=g.sim_duration))}')
        print(f'M Capacity: {self.m}')

        print(f'\nRequests made: {self.request_counter}')
        print(f'Requests lost: {self.lost_request_counter}')
        print(f'Memory full lost: {self.memory_full_loss}')
        print(f'Already Running lost: {self.already_running_loss}')
        print(f'Completions: {self.completions}')
        print(f'Cold starts: {self.cold_start_counter}')

        print(f'\nCold start ratio = {self.obs_cold_ratio[-1]}')
        print(f'Loss rate = {self.obs_loss_rate[-1]}\n')

        # Output to graphs
        ax.plot(model.obs_time, model.obs_cold_ratio, label=f'{self.m}')
        ax2.plot(model.obs_time, model.obs_loss_rate, label=f'{self.m}')

        # Output to csv
        with open(self.results_file, "a") as f:
            writer = csv.writer(f, delimiter=',')
            results_to_write = [self.trial_no,
                                self.obs_cold_ratio[-1],
                                self.obs_loss_rate[-1]]
            writer.writerow(results_to_write)


def generate_time_to_request():
    # aggregated poisson
    return np.random.exponential(scale=(1.0/g.sum_lam))

def generate_time_to_service(id):
    # exponential service time sampler
    return np.random.exponential(scale=(g.avg_service_times[id]))

def generate_time_to_start():
    # exponential cold start time sampler
    return np.random.exponential(scale=1.0/g.cold_start)

def remove_values_from_list(l, val):
    # removes all values = val from list l
    c = l.count(val)
    for i in range(c):
        l.remove(val)

    return l


# initialize graphs
fig = plt.figure(figsize=(12, 6))

ax = fig.add_subplot(121)
ax2 = fig.add_subplot(122)

# each loop here modifies the size of m
for run in range(g.no_runs):
```

```
    m = 40 + run * 5

    #creates a cdv file for each size of m simulated and writes coloumn headers
    file_name = f'sim_out/trial_results_{m}.csv'

    with open(file_name, "w") as f:
        writer = csv.writer(f, delimiter=',')
        coloumn_headers = ["run", "cold_start_ratio", "loss_rate"]
        writer.writerow(coloumn_headers)

    #repeats this capacity
    for trial in range(g.no_trials):
        model = Faas_Model(trial, m, file_name)
        model.run()

#finializes and shows graphs
ax.set_title('Cold Start Ratio')
ax2.set_title('Loss Rate')

ax.set_ylim([0, 0.3])
ax.set_xlim(left=0)

ax2.set_ylim([0, 100])
ax2.set_xlim(left=0)

plt.show()
```

A.2 Output Analysis

Below is the full code used to format the output of the simulation calculating key measures such as C_{ratio} and L_{rate}

```
import csv
import numpy as np
import matplotlib.pyplot as plt

def parse_csv(file):
    # grabs data from sim output

    with open(file, mode='r') as csv_file:
        csv_reader = csv.DictReader(csv_file)

        n = 0
        csr = []
        lr = []

        for row in csv_reader:
            csr.append(float(row["cold_start_ratio"]))
            lr.append(float(row["loss_rate"]))
            n += 1

    return n, csr, lr

ms = []
csr_points = []
lr_points = []
csr_ci = []
lower_csr_ci = []
higher_csr_ci = []

lr_ci = []
lower_lr_ci = []
higher_lr_ci = []

#create CSVs for storing point estimtes and confidence intervals for each vlue of m

with open('data/cold_start_data.csv', "w", newline='') as f:
    writer = csv.writer(f, delimiter=',')
    coloumn_headers = ["M", "Point Estimate", "90% Confidence Interval"]
    writer.writerow(coloumn_headers)

with open('data/loss_rate_data.csv', "w", newline='') as f:
    writer = csv.writer(f, delimiter=',')
    coloumn_headers = ["M", "Point Estimate", "90% Confidence Interval"]
    writer.writerow(coloumn_headers)
```

```

#read each output
for i in range(49):
  m = 40 + i*5
  ms.append(m)

  file_name = f'sim_out/trial_results_{m}.csv'

  n, csr, lr = parse_csv(file_name)

  csr_point_est = round(np.mean(csr), 4)
  csr_points.append(csr_point_est)

  lr_point_est = round(np.mean(lr), 4)
  lr_points.append(lr_point_est)

  #calculates point estimate and confidence interval for c_ratio
  csr_confidence_interval = round((1.833*np.std(csr))/np.sqrt(n), 4)
  csr_ci.append(csr_point_est)
  lower_csr_ci.append(csr_point_est-csr_confidence_interval)
  higher_csr_ci.append(csr_point_est+csr_confidence_interval)

    #calculates point estimate and confidence interval for l_rate
  lr_confidence_interval = round((1.833*np.std(lr))/np.sqrt(n), 4)
  lr_ci.append(lr_point_est)
  lower_lr_ci.append(lr_point_est-lr_confidence_interval)
  higher_lr_ci.append(lr_point_est+lr_confidence_interval)

  print(f'M Capacity: {m}')
  print(f'Samples: {n}\n')

  print(f'Cold Start Ratio:')
  print(f'Point Estimate: {csr_point_est:.4f}')
  ci = f'({csr_point_est-csr_confidence_interval:.4f}, {csr_point_est+csr_confidence_interval:.4f})'
  print(f'Confidence Interval: {ci}\n')

  with open('data/cold_start_data.csv', "a", newline='') as f:
    writer = csv.writer(f, delimiter=',')
    results_to_write = [m, f'{csr_point_est:.4f}', ci]
    writer.writerow(results_to_write)

  print(f'Loss Rate:')
  print(f'Point Estimate: {lr_point_est:.4f}')
  ci = f'({lr_point_est-lr_confidence_interval:.4f}, {lr_point_est+lr_confidence_interval:.4f})'
  print(f'Confidence Interval: {(lr_point_est-lr_confidence_interval):.4f}, {(lr_point_est+lr_confidence_interval):.4f}\n')

  with open('data/loss_rate_data.csv', "a", newline='') as f:
    writer = csv.writer(f, delimiter=',')
    results_to_write = [m, f'{lr_point_est:.4f}', ci]
    writer.writerow(results_to_write)

fig = plt.figure(figsize=(16, 8))

ax = fig.add_subplot(121)
ax2 = fig.add_subplot(122)

ax.set_title('Cold Start Ratio')
ax2.set_title('Loss Rate')

ax.plot(ms, csr_points)
ax.plot(ms, lower_csr_ci, color='red')
ax.plot(ms, higher_csr_ci, color='red')
ax.axhline(y=0.05, color='green')

ax2.plot(ms, lr_points)
ax2.plot(ms, lower_lr_ci, color='red')
ax2.plot(ms, higher_lr_ci, color='red')

ax.set_ylim([0, 0.17])
ax.set_xlim(left=0)

ax2.set_ylim([40, 100])
ax2.set_xlim(left=0)
```

```
plt.show()
```

B Results

B.1 Cold Start Ratio Data Varying M

M	Point Estimate	90% Confidence Interval
5	0.0175	(0.0158, 0.0192)
10	0.0358	(0.0354, 0.0362)
15	0.0540	(0.0521, 0.0559)
20	0.0735	(0.0718, 0.0752)
25	0.0909	(0.0888, 0.0930)
30	0.1104	(0.1090, 0.1118)
35	0.1266	(0.1235, 0.1297)
40	0.1426	(0.1412, 0.1440)
45	0.1577	(0.1562, 0.1592)
50	0.1646	(0.1629, 0.1663)
55	0.1629	(0.1611, 0.1647)
60	0.1570	(0.1559, 0.1581)
65	0.1468	(0.1441, 0.1495)
70	0.1404	(0.1388, 0.1420)
75	0.1310	(0.1274, 0.1346)
80	0.1247	(0.1235, 0.1259)
85	0.1172	(0.1154, 0.1190)
90	0.1136	(0.1114, 0.1158)
95	0.1099	(0.1082, 0.1116)
100	0.1036	(0.1024, 0.1048)
105	0.1009	(0.0997, 0.1021)
110	0.0956	(0.0936, 0.0976)
115	0.0906	(0.0890, 0.0922)
120	0.0877	(0.0858, 0.0896)
125	0.0851	(0.0826, 0.0876)
130	0.0814	(0.0786, 0.0842)
135	0.0804	(0.0790, 0.0818)
140	0.0778	(0.0768, 0.0788)
145	0.0746	(0.0727, 0.0765)
150	0.0721	(0.0712, 0.0730)
155	0.0700	(0.0682, 0.0718)
160	0.0677	(0.0660, 0.0694)
165	0.0676	(0.0661, 0.0691)
170	0.0650	(0.0638, 0.0662)
175	0.0622	(0.0611, 0.0633)
180	0.0611	(0.0599, 0.0623)
185	0.0586	(0.0569, 0.0603)
190	0.0583	(0.0574, 0.0592)
195	0.0561	(0.0543, 0.0579)
200	0.0561	(0.0556, 0.0566)
205	0.0544	(0.0536, 0.0552)
210	0.0531	(0.0518, 0.0544)
215	0.0526	(0.0515, 0.0537)
220	0.0520	(0.0505, 0.0535)
225	0.0508	(0.0503, 0.0513)
230	0.0495	(0.0487, 0.0503)
235	0.0475	(0.0467, 0.0483)
240	0.0470	(0.0458, 0.0482)
245	0.0470	(0.0459, 0.0481)

B.2 Loss Rate Data Varying M

M	Point Estimate	90% Confidence Interval
5	93.6856	(93.0467, 94.3245)
10	91.1806	(90.9515, 91.4097)
15	89.6749	(89.4617, 89.8881)
20	87.7405	(87.1199, 88.3611)
25	86.1625	(85.5256, 86.7994)
30	83.6936	(83.3656, 84.0216)
35	81.8676	(81.1517, 82.5835)
40	79.3631	(79.0700, 79.6562)
45	76.3572	(75.9604, 76.7540)
50	72.4308	(71.5926, 73.2690)
55	68.5625	(67.9686, 69.1564)
60	65.2207	(64.7839, 65.6575)
65	62.8829	(62.6129, 63.1529)
70	61.5926	(60.9079, 62.2773)
75	60.3371	(59.9879, 60.6863)
80	60.3980	(60.2341, 60.5619)
85	59.9719	(59.4054, 60.5384)
90	59.3672	(59.1650, 59.5694)
95	59.0983	(58.8472, 59.3494)
100	59.0107	(58.5693, 59.4521)
105	58.5773	(58.2342, 58.9204)
110	58.3097	(58.2313, 58.3881)
115	58.3278	(58.0936, 58.5620)
120	58.6722	(58.3561, 58.9883)
125	58.0789	(57.6215, 58.5363)
130	58.1886	(57.9470, 58.4302)
135	58.2615	(57.9761, 58.5469)
140	57.6649	(57.3389, 57.9909)
145	58.0281	(57.6912, 58.3650)
150	57.5933	(57.2737, 57.9129)
155	57.6609	(57.1932, 58.1286)
160	57.7405	(57.5057, 57.9753)
165	57.5512	(57.0856, 58.0168)
170	57.6147	(57.3572, 57.8722)
175	57.9217	(57.6247, 58.2187)
180	57.1893	(56.7768, 57.6018)
185	57.5211	(57.1645, 57.8777)
190	57.5579	(57.1333, 57.9825)
195	57.5445	(57.1851, 57.9039)
200	57.4455	(57.1244, 57.7666)
205	57.3933	(57.0981, 57.6885)
210	57.7298	(57.2964, 58.1632)
215	57.2495	(57.0320, 57.4670)
220	57.1385	(56.9153, 57.3617)
225	57.1204	(56.9562, 57.2846)
230	57.3806	(57.1006, 57.6606)
235	57.1980	(56.6960, 57.7000)
240	57.3284	(57.0875, 57.5693)
245	57.1826	(56.9856, 57.3796)