The Project Report for

"Choosing Optimal Cluster Configuration using Discrete Event Simulation"

Submitted for a course

MIE1613 – Stochastic Simulation

Submitted by

Vijaykumar Maraviya MEng. Student

(Student ID: 1006040320)

Guided by

Dr. Vahid Sarhangian Assistant Professor, MIE, U of T

Department of Mechanical and Industrial Engineering

University of Toronto

Table of Contents

1. Introduction

Cloud Computing has become a norm for the organizations' computational needs. Cloud computing provides ubiquitous access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) over the internet. Cloud Services offer faster innovation and flexible resources. The resources can be rapidly provisioned and released with minimal management effort or cloud provider interaction. By sharing the underlying hardware and infrastructure with multiple tenants and optimizing for the fluctuation in their need for resources, the cloud providers have achieved economies of scale, the benefits of which are passed on to the tenants. Still, the tenants of IaaS (Infrastructure as a Service) must make decisions when configuring the cloud infrastructure that will affect their bottom line. The cloud providers offer little assistance because the optimality of configurations vary from tenant to tenant. It depends on the nature of computing workloads and requirement for service guarantee tenants need to provide to their users. Often, the tenants make decisions based on their experience or the feedback from the community of practice. Choosing better configurations and policies in this manner requires a significant experimentation on a physical cluster over a long period of time. Despite, the chosen configuration may not be optimal.

In this report, a novel approach based on discrete event simulation has been described to aid in the decision making. First, a discrete event simulator is built to model a cloud computing cluster. The object-oriented paradigm is used to make the model extensible to simulate different type of clusters. Then, the use of the simulator to choose an optimal configuration for a specific (synthetic) workload for a certain scheduling policy is demonstrated. Finally, the synthesis of a creative schedular policy is shown to highlight the capability of this simulation-based approach. Overall, the proposed approach provides significant time reduction in choosing near optimal configurations for reducing the cost of operating a cluster in the cloud.

The report is organized as follows: first, a brief overview a relevant cloud concepts is given. In the third section the problem is described in more detail. In the fourth section, the modelling and analysis approach is explained. In the fifth section, the results are discussed. Finally, in the sixth and seventh section, the report is concluded with the remarks on the future scope of work in the direction of this approach.

2. Cloud Concepts

Deployment of large-scale distributed applications on Cloud have given rise to new architectures native to Cloud. Contrary to the traditional monolithic tightly coupled architectures, this new *micro-services*-based architecture is composed of several smaller, specialized processes that interact with each other to serve the users. These processes run isolated inside *containers* and can communicate with each other.

Containers are standalone, self-contained units. It packages all the software along with its dependencies required to run a process. In comparison to Virtual Machines (VMs), which provides virtualization at hardware level, containers provide virtualization at operating system (OS) level. It creates the illusion of the process being deployed on the isolated OS; despite, the processes are running on the same OS. Hence, the use of containers enables the sharing of resources of a single compute node between multiple users, processes, or applications in an isolated manner. Moreover, containers enable horizontal scaling of resources, workload partitioning, and modularity at a process, user, or application level. Verma et al. [1] provided the results on how the resource utilization improved by using containers at Google in terms of the number of machines needed to host a given workload on them.

Docker [2] and Linux Containers (LXC) [3] provide *a lightweight container run-time* to run the containers on a single machine in either private or in virtual public cloud environments. In practice, the containerized applications are deployed on a cluster of compute nodes. The containerized applications include web services, big data, IoT, High Performance Computing, and so on. The deployment of containers for these services requires container orchestration framework such as Kubernetes [4], Apache Mesos [5], and

Docker Swarm [6]. These frameworks manage the deployment, scaling, and management of various containers that constitute a service or application.

Kubernetes [4] is the most popular and open-source *container orchestration framework* used in industry today, which is derived from Google's in-house Borg [1]. Hence, it is used as a reference while designing the simulator. The key Kubernetes concepts and terminologies which are relevant for the project are described below:

Control plane, nodes, and node pools are core infrastructure components of Kubernetes cluster.

a) ***Control plane:*** It consists of components that are responsible for managing containerized application workloads on the worker nodes. For example, the API server, etcd, controller, and schedular are all part of control plane. All these components run on a master node and manages the workload of the cluster. Often, the resources required to run these components are managed by cloud provider and tenants are not charged for it.

b) ***Node:*** To run application containers, a tenant needs Kubernetes nodes, often referred to as worker nodes. A worker node can be a physical machine or a virtual machine. It is defined by the size and type of resources. Cloud service providers offer many different types of nodes with competitive pricing based on the amount and type of resources. The nodes are priced on an hourly basis and charged on second-by-second basis under pay-as-you-go pricing model.

   A Kubernetes node runs a few Kubernetes components required for control plane to coordinate with it such as *Kubelet* and *kubeproxy.* Additionally, It also runs *container runtime* (such as Docker) for containers to run and interact with node resources. All these components consume the node resources such as CPU and Memory; and hence, they are overhead for a node. The amount of resources they reserved are a significant proportion for a small node size. For nodes of bigger sizes, with additional amount of resources added to the node, only a small fraction of additional resources is reserved in addition to what a smaller node would reserve.

c) ***Node Pool:*** Nodes of the same types are grouped together into node pools. Kubernetes cluster contains at least one node pool with at least one node. The number of nodes (minimum and maximum) and node type for a pool are defined when a cluster is created. The number of active nodes inside a pool can *scale up* and *scale down* (***cluster autoscaling***) based on the demand of resources needed to run the *pods*. Multiple pools can be used to create complex policies that can achieve better overall utilization of the cluster, as demonstrated in the later sections.

d) ***Pod:*** A pod is a logical resource that encapsulates a container or containers representing a process, user, or micro-service application. Typically, pods have 1:1 mapping with a container. In some scenarios, it may contain multiple containers. Pods are usually ephemeral and disposable resources and represents resource demand and isolation of an underlying container. Multiple pods of same process in certain applications give the Kubernetes cluster *high availability* and *redundancy features*. A pod has a several attributes. Two important ones are: ***resource requests*** and ***resource limits***. Pods are scheduled to run on nodes by a schedular while ensuring that it can at least obtain resources defined in *resource requests*. A pod can utilize resources up to limit defined in *resource limits* if a node on which it is running have free resources available. This is referred to as ***Horizontal Pod Auto Scaling*** **(HPA)**. If a pod tries to use more Memory than defined in *resource limits*, it will be killed and removed. However, if it tries to utilize more CPU, it will be throttled but not killed.

e) ***Schedular:*** A schedular is a part of control plan, which is relevant to the discussion in subsequent sections. Its main purpose is to find a suitable node to run the pod. Kubernetes comes with built-in schedular. However, a custom plug-in schedular can also be used with Kubernetes cluster. The built-in schedular is referred as *kube-scheduler*. It selects a node for a pod in a 2-step operation: Filtering and Scoring. A filtering finds nodes where it is feasible to place a pod. For instance, it checks whether a candidate node has enough available resources to meet the demand specified by pod's *resource requests*. Then, the schedular assign scores to each node that survived filtering based on pre-defined criteria. Finally, the most suitable node is selected. A tenant can configure the predicate for filtering step and rules for scoring step for different types of pods, or they can write a completely custom schedular.

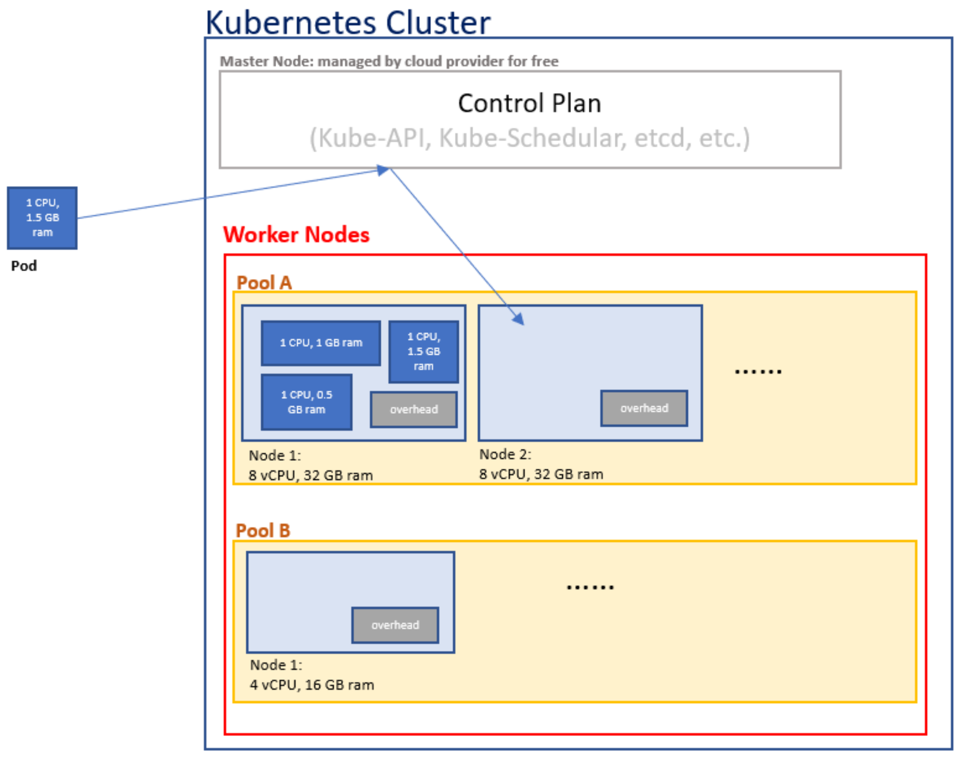

Kubernetes Cluster
Master Node: managed by cloud provider for free
Control Plan
(Kube-API, Kube-Schedular, etcd, etc.)
1 CPU, 1.5 GB ram
Pod
Worker Nodes
Pool A
1 CPU, 1 GB ram
1 CPU, 1.5 GB ram
1 CPU, 0.5 GB ram
overhead
overhead
......
Node 1: 8 vCPU, 32 GB ram
Node 2: 8 vCPU, 32 GB ram
Pool B
overhead
......
Node 1: 4 vCPU, 16 GB ram

Figure 1 Schematic diagram for Kubernetes cluster

3. Problem Description

When configuring a cloud infrastructure, a tenant must make choice about certain parameters. For instance, how many node pools should be used? What would be the optimum size and type of a node for each node pool? What scheduling policy should be used?

The choice of these parameters depends on the type of workload. Primarily, workload can be categorized into long running services and batch jobs. The long-running services often require high availability and must handle requests with low latency. They are often stateless, meaning that they can be killed and restarted without affecting other components. This type of workload is deployed using multiple redundant pods. Hence it is possible to kill some pods if It facilitates the better scheduling and resource utilization. User-facing web apps or services are example of this type of workload. The batch jobs have limited lifetime and are tolerable towards performance fluctuations. However, they cannot be interrupted while the jobs are being executed. Scientific computations or map-reduce jobs are example of batch jobs. In this project, only the batch jobs type workload has been simulated. However, the simulator has been designed to accommodate all kinds of workload with very little change in the code.

The synthetic batch jobs type workload has been used as an input to the system. A single batch job is considered to submitted as a single pod. For a batch job, the resource requests and limits can be estimated from historical data. There are three attributes that needs to be defined: arrivals (submission) of pods, runtime of a pod, and resourced required by the pod. The parameters of probability distribution used to generate these attributes are describe in the Analysis section.

Once the workload type and attributes are known, the tenants need to make optimal choices. For that, it is important to choose criteria to evaluate different choices. Most often, two criteria are used: 1) cost and 2) service guarantee. The cost of running a cluster for a unit time is used to compare various configurations. For service guarantee, the success probability of pods is used. The objective is to choose a configuration (node size (multiple node sizes if multiple pools are used) and schedular policy) such that the cost is minimized while providing the higher probability of success than defined.

In this project, this problem of choosing an optimal configuration is addressed in two steps. In the first step, the schedular policy is selected based on a heuristic and cluster's performance and cost is obtained using simulation for various node sizes. In the second step, once an optimal node size is found for a given workload and policy, a new configuration and policy is created using the insights generated from the simulation data of the first step. Manually synthesized configuration with multiple pools and custom schedular reduced the cost further.

4. Modelling

A Kubernetes type cluster is simulated using discrete event simulation in Python. The simulation framework `simpy` is used as base. It is further extended to create python classes for node, node pools, cluster, pod, pod generator and so on. Each class contains the attributes and methods to define behavior, calculate statistics, generate plots, etc. The summary is provided below.

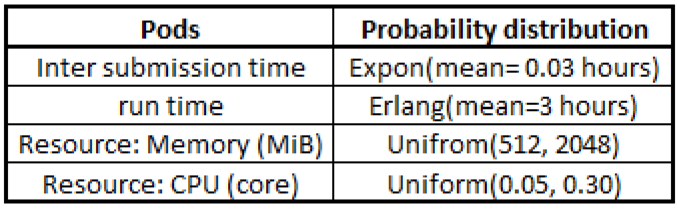
Primary, A pod generator process generates pod and submit it to the schedular. The schedular will access cluster state and allocate a node to the submitted pod. If there are no node that can fulfill the pod's resource request, a schedular can request the pool to add a new node. Once a new node is added, it is assigned to the pod. After a node is assigned to the pod, it can request resources from it. If node can fulfill the request, it will allocate requested resources to the pod and pod will run its process for a certain amount of time. Otherwise, it would fail. When pod terminate, either successfully or due to failure, it reports it back to the pod generator. All the request to acquire resources from a node, requests to add a node to the node pool, generation of pods, pod's submission and termination, and so on are modelled as events. The simulation clock will jump from event to event.

More details can be found in the code attached in the appendix.

5. Analysis

5.1 Workload

The time between submission of two consecutive pods is sampled from exponential distribution. This is justified because the arrival of pods represents the arrival of users in the workload that is used for simulation. Often, the arrival of users in queuing systems are simulated using Poisson process and hence, the exponential interarrivals. The run time of a pod is sampled from Erlang distribution (similar to service time of a queueing system). Usually, pod's memory and CPU usage vary in time during its runtime. To simplify the analysis and increase simulation speed, It is set as constant after being sample from uniform distribution. All the parameters are chosen based on experience from working with actual Kubernetes cluster, which are summarized in table below.

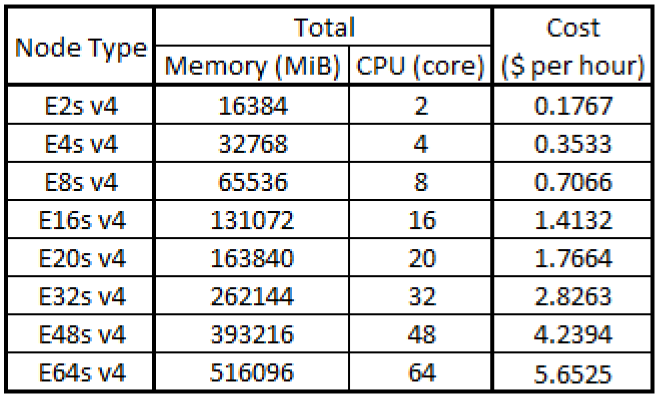


Pods	Probability distribution
Inter submission time	Expon(mean= 0.03 hours)
run time	Erlang(mean=3 hours)
Resource: Memory (MiB)	Unifrom(512, 2048)
Resource: CPU (core)	Uniform(0.05, 0.30)

Figure 2 Probability distributions Used to Sample Workload Parameters

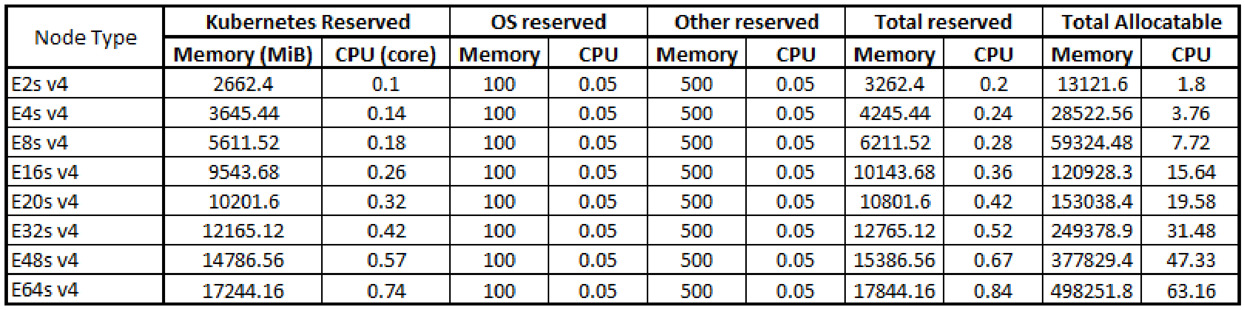
5.2 Node Size

Cloud service provider offers hundred of options to accommodate for all types of tenant needs. For the analysis performed in this report, the memory optimized node types are used based on the type of workload. The cost, overhead due to Kubernetes control plane process and total allocatable resources are summarized in the tables below. The offerings of Microsoft Azure are considered.



Node Type	Total		Cost
	Memory (MiB)	CPU (core)	($ per hour)
E2s v4	16384	2	0.1767
E4s v4	32768	4	0.3533
E8s v4	65536	8	0.7066
E16s v4	131072	16	1.4132
E20s v4	163840	20	1.7664
E32s v4	262144	32	2.8263
E48s v4	393216	48	4.2394
E64s v4	516096	64	5.6525

Figure 3 Memory Optimized Node Types Offered by Microsoft Azure



Node Type	Kubernetes Reserved		OS reserved		Other reserved		Total reserved		Total Allocatable	
	Memory (MiB)	CPU (core)	Memory	CPU	Memory	CPU	Memory	CPU	Memory	CPU
E2s v4	2662.4	0.1	100	0.05	500	0.05	3262.4	0.2	13121.6	1.8
E4s v4	3645.44	0.14	100	0.05	500	0.05	4245.44	0.24	28522.56	3.76
E8s v4	5611.52	0.18	100	0.05	500	0.05	6211.52	0.28	59324.48	7.72
E16s v4	9543.68	0.26	100	0.05	500	0.05	10143.68	0.36	120928.3	15.64
E20s v4	10201.6	0.32	100	0.05	500	0.05	10801.6	0.42	153038.4	19.58
E32s v4	12165.12	0.42	100	0.05	500	0.05	12765.12	0.52	249378.9	31.48
E48s v4	14786.56	0.57	100	0.05	500	0.05	15386.56	0.67	377829.4	47.33
E64s v4	17244.16	0.74	100	0.05	500	0.05	17844.16	0.84	498251.8	63.16

Figure 4 Resource Allocated for Overhead Processes and Total Allocatable Resources

5.3 Schedular Policy

To simulate workload defined above using the node sizes mentioned, a best fitting bin packing scheduling algorithm is implemented. This algorithm assigns new pod to the most utilized node in the cluster if it is possible to do so. If there are no suitable nodes that can fulfill the resource guarantees of a pod, the schedular will ask the pool with smallest node size to add a new node and assign that to the pod. This algorithm favors the better scale-in and resource utilization of nodes and reduces cost by effectively removing nodes that are not being used.

6. Results

For all node sizes mentioned in the previous section, the simulation is run for 100 hours to obtain estimate of pod success probability and cluster's unit cost (cost of running a cluster for 1 hour). These results are used for two purposes. First, it is observed that it requires some time for cluster to achieve steady state. Hence, the conservative approach is used to estimate the deletion point. Ideally, all the different cluster configuration achieves steady state at different times. However, the maximum of all the configuration deletion points is used conservatively because it is cheap to run simulation for more time. The plots for 3 different representative node sizes are shown below.

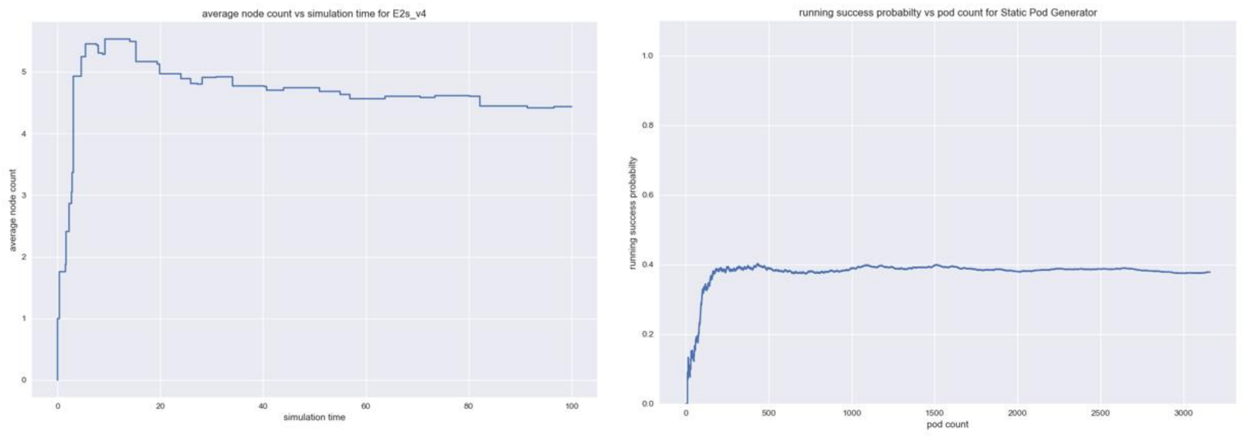


Figure 5 Running Averages for node count and success probability for node type E2s_v4

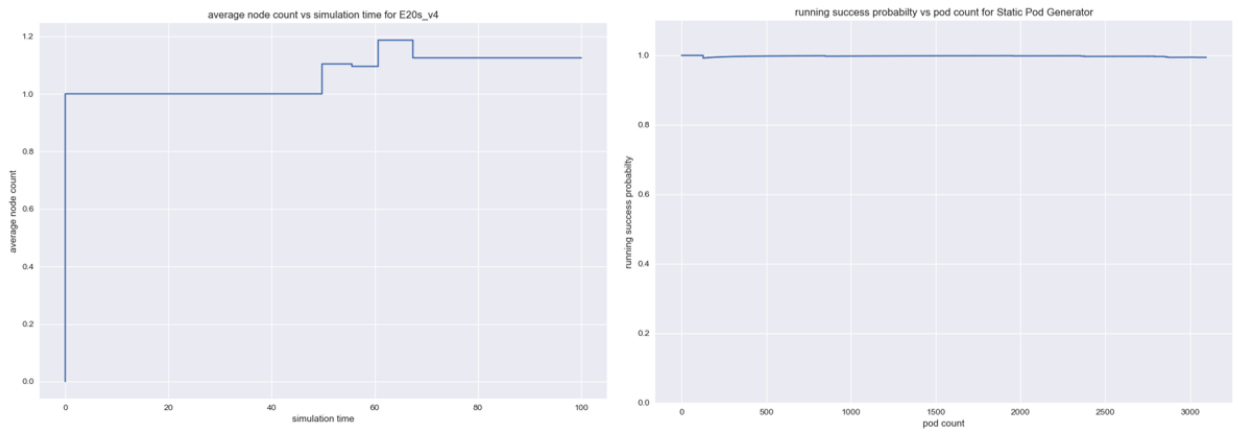


Figure 6 Running Averages for node count and success probability for node type E20s_v4

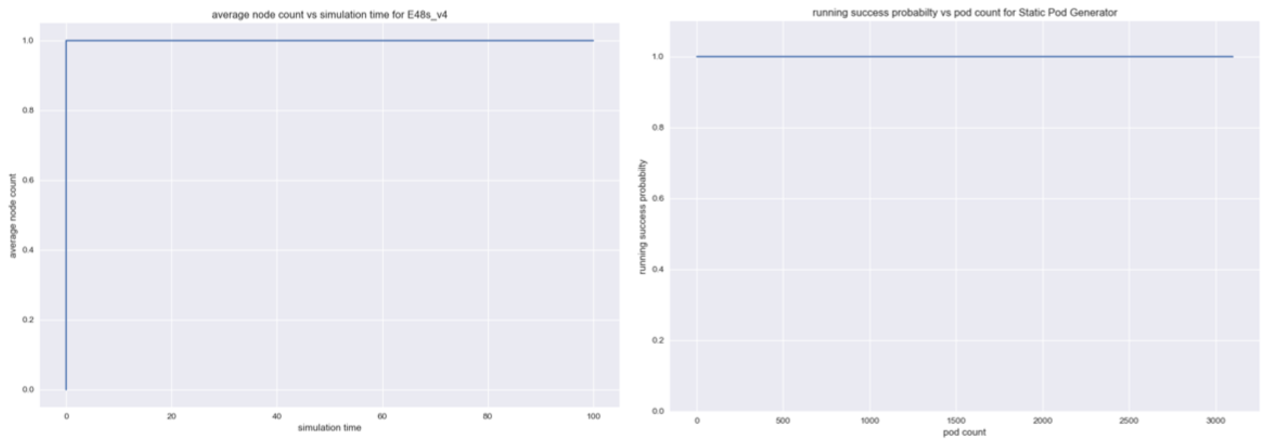


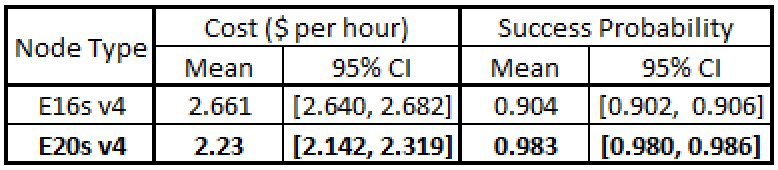
Figure 7 Running Averages for node count and success probability for node type E48s_v4

Above graph shows the steady state behavior. There is a bias in the estimate at the beginning of the simulation due to zero start of the cluster. However, the bias quickly diminishes with time. It can be observed that for a smallest node size success probability is less than 40%, where medium size it is more than 90%. The table below summarizes the results of simulations for all node sizes. The node types marked in the red violets the criteria of success probability (required >90%). The node types marked in blue are the one that have probability of almost 100%. However, they are under-utilized and hence the cost is higher. From this part of analysis node sizes 'E16s_v4' and 'E20s_v4' are chosen for further analysis.

Node Type	Cost ($ per hour)	Success Probability
E2s v4	0.784	0.378
E4s v4	1.087	0.530
E8s v4	1.399	0.684
E16s v4	**2.474**	**0.907**
E20s v4	**1.988**	**0.994**
E32s v4	2.8263	0.999
E48s v4	4.2394	0.999
E64s v4	5.6525	0.999

Figure 8 Estimated cost and success probability for each node size

Subsequently, two chosen nodes sizes are simulated for 10 runs, each of length 500 hours, to obtain estimate with tighter confidence intervals, and the results are reported in the table below.

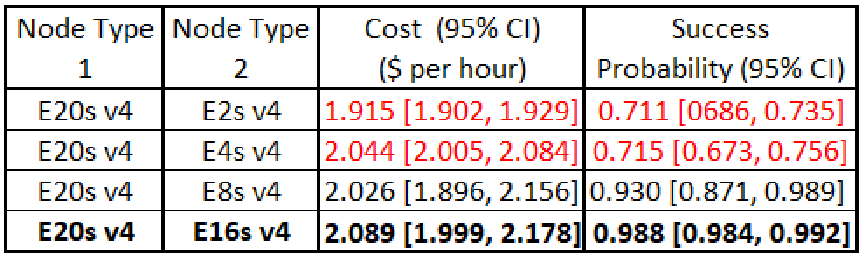


Node Type	Cost ($ per hour)		Success Probability	
	Mean	95% CI	Mean	95% CI
E16s v4	2.661	[2.640, 2.682]	0.904	[0.902, 0.906]
E20s v4	**2.23**	**[2.142, 2.319]**	**0.983**	**[0.980, 0.986]**

Figure 9 95% CI of cost and success probability for selected node sizes

Finally, node size E20s_v4 is selected. In comparison to node size (E32s_v4) used by ITS, U of T to run a cluster with similar workload, the cost is 21% less.

Furthermore, in the 2nd step of analysis, two pools are used. The primary pool has a node size defined by E20s_v4. For a node size of second pool, E2s_v4, E4s_v4, E8s_v4, and E16s_v4 are simulated along with first pool of size E20s_v4. With best fit bin packing schedular, the second pool with node type E16s_v4 yielded the optimal results.



Node Type 1	Node Type 2	Cost (95% CI) ($ per hour)	Success Probability (95% CI)
E20s v4	E2s v4	1.915 [1.902, 1.929]	0.711 [0686, 0.735]
E20s v4	E4s v4	2.044 [2.005, 2.084]	0.715 [0.673, 0.756]
E20s v4	E8s v4	2.026 [1.896, 2.156]	0.930 [0.871, 0.989]
E20s v4	**E16s v4**	**2.089 [1.999, 2.178]**	**0.988 [0.984, 0.992]**

Figure 10 Multi Pool configuration to achieve better optimality

The above results can be explained by looking at the memory utilization graphs of nodes for each configuration. Once a smaller node is added to the Pool B, the schedular always choose the node of pool B in the case of first (E20s_v4 + E2s_v4) and second configuration (E20s_v4 + E4s_v4). This is because the smaller are more utilized as soon as they are added to the pool because of overhead. Hence, the utilization of nodes in the first pool reduces over time, while schedular keeps adding nodes to the second pool. This leads to lower success probability over time as reflected in the results reported above. For third and fourth configurations, mean success probability is higher than defined threshold of 90%. However, the variance of estimate is higher for 3rd configuration (E20s_v4 + E8s_v4), and it may go below 90% for a period time. Hence, It is also discarded despite lower cost compared to fourth configuration.

Finally, by comparing fourth configuration (E20s_v4 + E16s_v4) with optimal configuration of 1st step (E20s_v4), It can be seen that the cost is reduced by 15% while probability has improved slightly. This improvement is attributed to the fact that the node E20s_v4 only rarely exceeds the capacity and requires scale-out operation to add a new node. Hence, if a smaller node is added that do not get fully utilized with a very few pods, the cost can be improved. This is because the schedular will keep scheduling pods on main pool if possible and only use 2nd pool to avoid adding a node to the main pool with large size. Hence, this shows that, by using multiple pool along with information about workload, the cost can be improved with custom policies and configurations.

7. Conclusion

In this project, a cloud simulator has been developed that can be used for finding optimal configurations and policies. The use of simulator has been shown using a specific workload of batch jobs type and memory optimized node sizes. The best fit bin packing schedular is used to minimize the cost. The simulator has been designed such that it can be easily extended to other workloads, node sizes, and schedular policies. It is imperative that the simulator is tested using workload with parameters estimated from real world data. Similarly, many different scheduling policies for different cluster configurations under different workloads can be tested using the approach demonstrated in this report.

8. References

[1] Verma A, Pedrosa L, Korupolu M, Oppenheimer D, Tune E, Wilkes J. Large scale cluster management at Google with Borg. In Proceedings of the 10th European Conference on Computer Systems (EuroSys 2015), Bordeaux, France — April 21 - 24, 2015. p. 18.
[2] Docker. https://www.docker.com. Accessed on May2021.
[3] D. Bernstein, Containers and Cloud: From LXC to Docker to Kubernetes, IEEE Cloud Computing, vol. 1, no. 3, pp. 81-84, 2014.
[4] Hightower K, Burns B, Beda J, Kubernetes: Up and Running: Dive Into the Future of Infrastructure, O'Reilly Media, Inc., 2017.
[5] N. Naik, Building a virtual system of systems using docker swarm in multiple clouds, In Proceedings of the 2016 IEEE International Symposium on Systems Engineering (ISSE), Edinburgh, 2016, pp. 1-3.
[6] Hindman B, Konwinski A, Zaharia M, Ghodsi A, Joseph AD, Katz RH, et al. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI); 2011. p. 22–22.

9. Appendix

Code:

Pod.py

```python
import random
import logging

# custom classes
from utils import Resource
from Node import Node

# setup logger
logger = logging.getLogger(__name__)

logger.setLevel(logging.DEBUG)
formatter = logging.Formatter('%(levelname)s:%(message)s')

file_handler = logging.FileHandler('Simulation.log')
file_handler.setFormatter(formatter)

logger.addHandler(file_handler)


class Pod:
    '''
    Base class for pod.
    Subclass this and implement `process()` method to define pod behaviour.
    '''

    def __init__(self, env, name, resource_guarantee, resource_limit):
        '''
        `resource_limit` is a NamedTuple(memory, cpu) max limits.

        '''
        self._env = env
        self._name = name

        # pod resource gaurantees
        self._mem_guarantee = resource_guarantee.memory
        self._cpu_guarantee = resource_guarantee.cpu
        # pod resource limits
        self._mem_limit = resource_limit.memory
        self._cpu_limit = resource_limit.cpu
        # pod resource usage
        self._mem_usage = 0
        self._cpu_usage = 0
        # pod resource max usage
        self._max_mem_usage = 0
        self._max_cpu_usage = 0

        # pod attributes (set at runtime)
        self._create_time = None  # set by pod generator
        self._start_time = None  # set by pod `.process()`
        self._finish_time = None  # set by pod `.process()`
        self._success = None  # set by pod `.process()`
        # message when pod terminates (use to troublshoot pod failures)
        self._value = None  # set by pod `.process()`

        # Node assigned to pod
        self._node = None

    @property
    def name(self):
        return self._name

    @property
    def mem_guarantee(self):
        return self._mem_guarantee

    @property
    def cpu_guarantee(self):
        return self._cpu_guarantee

    @property
    def mem_limit(self):
        return self._mem_limit

    @property
    def cpu_limit(self):
        return self._cpu_limit

    @property
    def mem_usage(self):
        return self._mem_usage

    @property
    def cpu_usage(self):
        return self._cpu_usage
```

```python
@property
def max_mem_usage(self):
    return self._max_mem_usage

@property
def max_cpu_usage(self):
    return self._max_cpu_usage

@property
def create_time(self):
    return self._create_time

@create_time.setter
def create_time(self, time):
    # TODO: check input before assignment
    if self._create_time is None:
        self._create_time = time
    else:
        raise ValueError('Pod already submitted.')

@property
def start_time(self):
    return self._start_time

@property
def finish_time(self):
    return self._finish_time

@property
def success(self):
    return self._success

@property
def node(self):
    return self._node

@node.setter
def node(self, node):
    if isinstance(node, Node):
        self._node = node
    else:
        raise ValueError("Invalid object type.")

@property
```

```
    def value(self):
        return self._value

    def process(self, *args, **kwargs):
        '''
        Define the beahviour of a pod while running on an assigned node.

        '''
        raise NotImplementedError


class StaticPod(Pod):
    def __init__(self, env, name, resource_guarantee, resource_limit):
        super().__init__(env, name, resource_guarantee, resource_limit)

    def process(self, period_gen, resource_use_gen, schedular):
        '''
        `period_gen` is python generator which yields the time needed for pod to
finish the job.
        `resource_use_gen` is python generator which yields the nametuple of reso
urce used by the pod.
        `schedular` is the instance of Schedular Base class and allocates the nod
e.
        '''
        # checks to ensure that one pod runs only once
        if self._success is not None:
            raise ValueError('Pod is already processed')

        try:

            with schedular.allocate(self) as res:
                # resume when node is allocated
                yield res

            # DEBUG: which node is asigned
            logger.debug(
                f"Time= {self._env.now} | {self.name} is assigned {self.node.ID}"
)

            # resources needed for pod to run
            mem_use, cpu_use = next(resource_use_gen)

            # acquire the resources from the assigned node
            with self._node.acquire(self, mem_use, cpu_use) as res:
                # resume when resources are obtained
                yield res

            # update pod attributes
            self._start_time = self._env.now
            self._mem_usage = mem_use
            self._cpu_usage = cpu_use
            self._max_mem_usage = mem_use
            self._max_cpu_usage = cpu_use

            # DEBUG: When and how much resources are obtained
            logger.debug(
                f"Time= {self._env.now} | {self.name} obtained RAM={mem_use} and
CPU={cpu_use}")

            # generate the pod run time
            run_time = next(period_gen)
            # resume when pod has finished
            yield self._env.timeout(run_time)

            # release the resources back to the node
            with self._node.release(self, mem_use, cpu_use) as res:
                yield res

            self._mem_usage = 0
            self._cpu_usage = 0

            # update the finish time
            self._finish_time = self._env.now

            # DEBUG: When and how much resources are released
            logger.debug(
                f"Time= {self._env.now} | {self.name} released the resources and
terminated")

            # mark success
            self._success = True

        except Exception as e:
            # DEBUG : When and why a pod is killed
            logger.debug(
                f"Time= {self._env.now} | {self.name} killed due to {e}")

            # mark failure
            self._success = False
            self._value = e
```

PodGenerator.py

```python
import matplotlib.pyplot as plt
import csv
import os
import itertools
import logging
from collections import namedtuple

# custom classes
from SimStatistics import CTStat, DTStat
from Pod import StaticPod
from utils import NodeMatchNotFound, change_dir

# simpy classes
import simpy.core as core

# setup logger
logger = logging.getLogger(__name__)

logger.setLevel(logging.DEBUG)
formatter = logging.Formatter('%(levelname)s:%(message)s')

file_handler = logging.FileHandler('Simulation.log')
file_handler.setFormatter(formatter)

logger.addHandler(file_handler)

# setup plot style
plt.style.use('seaborn')


class PodGenerator:
    # value of indicator function for pod sucess/failure event
    SUCCESS = 1
    FAILURE = 0

    def __init__(self, env, name, arrival_gen):
        self._env = env
        # arrival_gen yields interarrival time when next method is called
        self.t_gen = arrival_gen
        self._name = name
        # statistics (success probability)
        self._success_metric = DTStat(env, env.now)
        self._success_prob = None

    # TODO: add properties

    @property
    def success_prob(self):
        return self._success_prob

    def generate(self, resource_guarantee,
                 resource_limit, period_gen, resource_gen,
                 schedular):

        # TODO: user should provide pod class, change it.
        # TODO: user should provide schedular object, change it
        for i in itertools.count(1):

            # create a new pod
            pod = StaticPod(self._env, 'Pod %d' % i,
                            resource_guarantee, resource_limit)

            # set the create_time
            pod.create_time = self._env.now

            # DEBUG: When a pod is created
            logger.debug(
                f"Time= {pod.create_time} | {pod.name} is created")

            # start the pod process
            process = self._env.process(
                pod.process(period_gen, resource_gen, schedular))

            # callback to report the result back to the generator
            process.callbacks.append(self._result)
            process.pod = pod

            # resume after interarrival time to submit next pod
            yield self._env.timeout(next(self.t_gen))

    def _result(self, process):
        '''
        records the pod success/failures
        '''
        if process.pod.success:
            # record value of indicator function (1 for success)
            self._success_metric.Record(PodGenerator.SUCCESS)

        else:
            # record value of indicator function (0 for failure)
            self._success_metric.Record(PodGenerator.FAILURE)

        self.record()

    def record(self):
        # calculate the results
        self._success_prob = self._success_metric.Mean()

    def clear(self):
        self._success_metric.Clear()
        self.record()

    def get_results(self):
        self.record()

        PodResult = namedtuple('PodResults', ['success_probability'])

        results = PodResult(self._success_prob)

        return {self._name: results}

    def save_results(self, destination=None):
        # record statistics
        self._success_metric.Record(0)

        if destination is None:
            destination = os.getcwd()

        # inside destination directiory
        with change_dir(destination):
            pod_gen_dir = self._name
            # create a directory for pod generator
            os.mkdir(pod_gen_dir)

            # inside the pod gen directory
            with change_dir(pod_gen_dir):

                # data (metrics)
                sim_time = self._success_metric.T_list
                indicator_vals = self._success_metric.X_list
                success_prob_r_avg = self._success_metric.RunningAvg
                n_pods = range(1, len(indicator_vals) + 1)

                row_list = zip(sim_time, indicator_vals, success_prob_r_avg)

                target_file = self._name + '-Metrics.csv'

                with open(target_file, 'w', newline='') as new_file:

                    fieldnames = ['sim_time', 'success_ind_val',
                                  'success_prob_running_avg']

                    csv_writer = csv.writer(new_file)

                    csv_writer.writerow(fieldnames)
                    for row in row_list:
                        csv_writer.writerow(row)

                # create plots directory
                os.mkdir('plots')

                # inside plots directory
                with change_dir('plots'):
                    # save plots
                    # self._plot_and_save(sim_time, indicator_vals,
                    #                     'simulation time', 'success', plots_pat
h)
                    self._plot_and_save(sim_time, success_prob_r_avg,
                                        'simulation time', 'running success proba
bilty')
                    self._plot_and_save(n_pods, success_prob_r_avg,
                                        'pod count', 'running success probabilty'
)

    def _plot_and_save(self, x, y, xlabel, ylabel, destination=None):
        if destination is None:
            destination = os.getcwd()

        # new figure and associated axes
        fig1 = plt.figure(figsize=(12, 8))
        ax1 = fig1.add_subplot()

        # plot
        ax1.step(x, y, where='post')
        # set labels and titles
        ax1.set_xlabel(xlabel)
        ax1.set_ylabel(ylabel)
        ax1.set_ylim(0, 1.1)
        ax1.set_title(f'{ylabel} vs {xlabel} for {self._name}')

        # save file at destination directory
        fig1.savefig(os.path.join(
            destination, xlabel + '_vs_' + ylabel + '.png'))

        # close the figure
        plt.close(fig1)
```

Schedular.py

```
import logging

# simpy classes
from simpy.events import Event
from simpy.core import BoundClass, Environment

# custom classes
from Pod import Resource
from utils import NodeMatchNotFound, PoolMaxCapacityError

# setup logger
logger = logging.getLogger(__name__)

logger.setLevel(logging.DEBUG)
formatter = logging.Formatter('%(levelname)s:%(message)s')

file_handler = logging.FileHandler('Simulation.log')
file_handler.setFormatter(formatter)

logger.addHandler(file_handler)


class Allocate(Event):
    ''' Event for requesting to allocate a node
    '''

    def __init__(self, schedular, pod):
        super().__init__(schedular._env)
        self.resource = schedular
        self.pod = pod
        self.proc = self.env.active_process

        schedular.put_queue.append(self)
        schedular._do_allocate(self)

    def __enter__(self):
        return self

    def __exit__(self,
                 exc_type,
                 exc_value,
                 traceback):
        self.cancel()
        return None

    def cancel(self):
        """Cancel this put request.

        This method has to be called if the put request must be aborted, for
        example if a process needs to handle an exception like an
        :class:`~simpy.exceptions.Interrupt`.

        If the put request was created in a :keyword:`with` statement, this
        method is called automatically.

        """
        if not self.triggered:
            self.resource.put_queue.remove(self)


class Schedular:

    PutQueue = list

    def __init__(self, env, name, cluster):
        self._env = env
        self._name = name
        self._cluster = cluster

        # request queues
        self.put_queue = self.PutQueue()

        # Bind event constructors as methods
        BoundClass.bind_early(self)
    # put a node into the pool
    allocate = BoundClass(Allocate)

    @property
    def name(self):
        return self._name

    @property
    def cluster(self):
        return self._cluster

    def _do_allocate(self, event):
        '''
        This method is called when a pod rquest to allocate (allocate event) the
pod.
        Imlement the processing of allocation request.
        '''
        raise NotImplementedError


class BestFitBinPackingSchedular(Schedular):
    def __init__(self, env, name, cluster):
        super().__init__(env, name, cluster)

    def _do_allocate(self, event):
        """Perform the *put* operation.

        This method is called by :meth:`_trigger_put` for every event in the
        :attr:`put_queue`, as long as the return value does not evaluate
        ``False``.
        """

        demand = Resource(event.pod.mem_guarantee, event.pod.cpu_guarantee)
        try:
            assigned_node = self._cluster.get_node(
                demand, criteria='most_utilized')
            event.pod.node = assigned_node

            # succeed the request
            event._ok = True
            event._value = None
            # schedule the aloocation event immeadiately
            # TODO: logic is incorrect for delay start
            event.env.schedule(event, priority=0)
            # remove the event from the schedular queue
            self.put_queue.remove(event)

            # record the statistic
            # TODO: add record for statistics
            # self._node_metric.Record(self.node_count)

        except NodeMatchNotFound:

            self._env.process(self._attempt_to_scaleout(event, demand))

    def _attempt_to_scaleout(self, event, demand):
        # get nodepools and sort by size
        nodepools = sorted(self._cluster.pools,
                           key=lambda pool: pool.node_size)

        for nodepool in nodepools:
            try:
                # request to scaleout a pool
                with nodepool.scaleout() as req:
                    new_node = yield req

                # DEBUG: When and why a new node is added
                logger.debug(
                    f"Time= {self._env.now} | {new_node.ID} added while allocatin
g {event.pod.name}")

                # # if request succeed, assign the new node to the pod
                # assigned_node = nodepool.get_least_utilized_node(demand)
                # TODO: check if works?
                event.pod.node = new_node

                # succeed the request
                event._ok = True
                event._value = None
                # schedule immeadiately
                # TODO: logic is incorrect for delay start
                event.env.schedule(event, priority=0)
                # remove the event from the schedular queue
                self.put_queue.remove(event)

                return None

            except PoolMaxCapacityError:
                # DEBUG
                logger.debug(
                    f"Time= {self._env.now} | failed to add a node to {nodepool.n
ame}")

                # try with next nodepool
                pass

        # TODO: provide more informative error message
        # reject the request
        event._ok = False
        event._value = NodeMatchNotFound('Cannot allocate this pod!')
        event.env.schedule(event, 0)  # schedule immeadiately
        # remove the event from the schedular queue
        self.put_queue.remove(event)


class CustomSchedular(Schedular):
    def __init__(self, env, name, cluster):
        super().__init__(env, name, cluster)

    def _do_allocate(self, event):
        """Perform the *put* operation.

        This method is called by :meth:`_trigger_put` for every event in the
        :attr:`put_queue`, as long as the return value does not evaluate
        ``False``.
        """

        demand = Resource(event.pod.mem_guarantee, event.pod.cpu_guarantee)
        try:
            assigned_node = self._cluster.get_node(
                demand, criteria='most_utilized')
            event.pod.node = assigned_node

            # succeed the request
            event._ok = True
            event._value = None
            # schedule the aloocation event immeadiately
            # TODO: logic is incorrect for delay start
            event.env.schedule(event, priority=0)
            # remove the event from the schedular queue
            self.put_queue.remove(event)

            # record the statistic
            # TODO: add record for statistics
            # self._node_metric.Record(self.node_count)

        except NodeMatchNotFound:

            self._env.process(self._attempt_to_scaleout(event, demand))

    def _attempt_to_scaleout(self, event, demand):
        # get nodepools and sort by size
        nodepools = sorted(self._cluster.pools,
                           key=lambda pool: pool.node_size)

        for nodepool in nodepools:
            try:
                # request to scaleout a pool
                with nodepool.scaleout() as req:
                    new_node = yield req

                # DEBUG: When and why a new node is added
                logger.debug(
                    f"Time= {self._env.now} | {new_node.ID} added while allocatin
g {event.pod.name}")

                # # if request succeed, assign the new node to the pod
                # assigned_node = nodepool.get_least_utilized_node(demand)
                # TODO: check if works?
                event.pod.node = new_node

                # succeed the request
                event._ok = True
                event._value = None
                # schedule immeadiately
                # TODO: logic is incorrect for delay start
                event.env.schedule(event, priority=0)
                # remove the event from the schedular queue
                self.put_queue.remove(event)

                return None

            except PoolMaxCapacityError:

                # DEBUG
                logger.debug(
                    f"Time= {self._env.now} | failed to add a node to {nodepool.n
ame}")
```

```
                # try with next nodepool
                pass

        # TODO: provide more informative error message
        # reject the request
        event._ok = False
        event._value = NodeMatchNotFound('Cannot allocate this pod!')
        event.env.schedule(event, 0)  # schedule immeadiately
        # remove the event from the schedular queue
        self.put_queue.remove(event)
```

Node.py

```
import os
import csv
import matplotlib.pyplot as plt
from collections import namedtuple

# custom classes
from SimStatistics import CTStat, DTStat
from utils import (PoolMinCapacityError,
                   NodeOOMError,
                   NodeCPUError,
                   change_dir)

# simpy classes
from simpy.core import BoundClass, Environment
from simpy.resources import base
from simpy.exceptions import SimPyException


class NodeRelease(base.Put):
    '''Event for requesting to release compute resources

    Raise a `ValueError` if ``resources <= 0``.
    '''

    def __init__(self, node, pod, memory, CPU):
        if CPU < 0:
            raise ValueError(f'CPU(={CPU}) must be >=0.')
        if memory < 0:
            raise ValueError(f'memory(={memory}) must be >=0.')
        self.pod = pod

        self.CPU = CPU
        self.memory = memory

        super().__init__(node)


class NodeAcquire(base.Get):
    '''Event for requesting to aquire compute resources

    Raise a `ValueError` if ``resources <= 0``.
    '''

    def __init__(self, node, pod, memory, CPU):
        if CPU < 0:
            raise ValueError(f'CPU(={CPU}) must be >=0.')
        if memory < 0:
            raise ValueError(f'memory(={memory}) must be >=0.')

        self.pod = pod

        self.CPU = CPU
        self.memory = memory

        super().__init__(node)


class Node:
    ''' Node containing up to capacity of compute resources.
    It supports the request to acquire and release resources from/into the Node.

    Raise a `ValueError` if resource capacity <= 0, init < 0, or init > capacity
    '''

    PutQueue = list
    GetQueue = list

    def __init__(
            self,
            env,
            Node_ID,
            CPU_capacity,
            memory_capacity,
            CPU_init=0,
```

```python
            memory_init=0,
            pool=None,):
        if CPU_capacity <= 0:
            raise ValueError('"CPU_capacity" must be > 0.')
        if CPU_init < 0:
            raise ValueError('"CPU_init" must be >= 0.')
        if CPU_init > CPU_capacity:
            raise ValueError('"CPU_init" must be <= "CPU_capacity".')
        if memory_capacity <= 0:
            raise ValueError('"memory_capacity" must be > 0.')
        if memory_init < 0:
            raise ValueError('"memory_init" must be >= 0.')
        if memory_init > memory_capacity:
            raise ValueError('"memory_init" must be <= "memory_capacity".')

        # unique id to identify the node (provided by NodePool)
        self._Node_ID = Node_ID
        # nodepool to which node belong, if any
        self._nodepool = pool
        self._env = env
        self._bootup_time = env.now
        self._shutdown_time = float('inf')
        self._pods = set()

        # Node capacity
        self._CPU_capacity = CPU_capacity
        self._memory_capacity = memory_capacity

        # Available CPU and memory
        self._CPU_avail = CPU_capacity - CPU_init
        self._memory_avail = memory_capacity - memory_init

        # request queues
        self.put_queue = self.PutQueue()
        self.get_queue = self.GetQueue()

        # Bind event constructors as methods
        BoundClass.bind_early(self)

        # statistics (Node Metrics)
        # memory usage over time
        self._mem_metric = CTStat(env, env.now, memory_init)
        # cpu usage over time
        self._cpu_metric = CTStat(env, env.now, CPU_init)
        # pod count over time
        self._pod_metric = CTStat(env, env.now, 0)

        # result (summary statistics)
        self._mem_results = None
        self._cpu_results = None
        self._pod_results = None

    @property
    def ID(self):
        return self._Node_ID

    @property
    def nodepool(self):
        return self._nodepool

    @property
    def bootup_time(self):
        return self._bootup_time

    @property
    def shutdown_time(self):
        return self._shutdown_time

    @shutdown_time.setter
    def shutdown_time(self, time):
        '''
        shutdown time is set by NodePool
        '''
        if time >= self._env.now:
            self._shutdown_time = time
        else:
            raise ValueError("Invalid time")

    @property
    def pod_count(self):
        return len(self._pods)

    @property
    def pods(self):
        return self._pods

    @property
    def CPU_avail(self):
        """The current available CPU in the node."""
        return self._CPU_avail
    @property
    def memory_avail(self):
        """The current available memeroy in the node."""
        return self._memory_avail

    @property
    def CPU_capacity(self):
        """Maximum CPU capacity of the node."""
        return self._CPU_capacity

    @property
    def memory_capacity(self):
        """Maximum memory capacity of the node."""
        return self._memory_capacity

    # put resources into the node
    release = BoundClass(NodeRelease)
    # get resources from the node
    acquire = BoundClass(NodeAcquire)

    def __lt__(self, other):
        return (self.memory_avail, self.CPU_avail, self.bootup_time) < (other.mem
ory_avail, other.CPU_avail, other.bootup_time)

    def _nodepool_heap_update(self):
        '''
        update the position of node in nodepool heap after any state change
        '''
        self._nodepool.heap_update()

    def _do_put(self, event):
        """Perform the *put* operation.

        This method is called by :meth:`_trigger_put` for every event in the
        :attr:`put_queue`, as long as the return value does not evaluate
        ``False``.
        """
        if (self._CPU_capacity - self._CPU_avail >= event.CPU) and (self._memory_
capacity - self._memory_avail >= event.memory):
            # add the resources back
            self._CPU_avail += event.CPU
            self._memory_avail += event.memory

            if event.pod.mem_usage - event.memory == 0:
                # remove requester pod from pods list
                self._pods.remove(event.pod)

                # (autoscalling) if pod_count is zero, ask pool to scale in
                if self.pod_count == 0:
                    try:
                        self._nodepool.scalein()
                    except PoolMinCapacityError:
                        pass

            # update node position in the nodepool heap
            if self._nodepool is not None:
                self._nodepool_heap_update()

            # succeed the request
            # event.succeed()
            event._ok = True
            event._value = None
            event.env.schedule(event, 0)  # schedule immeadiately

            # record the node statistic
            self.record()

            return True

        else:
            # TODO: should raise an exception
            return None

    def _trigger_put(self, get_event):
        """This method is called once a new put(release) event has been created o
r a get(acquire)
        event has been processed.

        The method iterates over all put events in the :attr:`put_queue` and
        calls :meth:`_do_put` to check if the conditions for the event are met.
        If :meth:`_do_put` returns ``False``, the iteration is stopped early.
        """
        idx = 0
        while idx < len(self.put_queue):
            put_event = self.put_queue[idx]
            proceed = self._do_put(put_event)
            if not put_event.triggered:
                idx += 1
            elif self.put_queue.pop(idx) != put_event:
                raise RuntimeError('Put queue invariant violated')

            if not proceed:
                break

    def _do_get(self, event):
        """Perform the *get* operation.

        This method is called by :meth:`_trigger_get` for every event in the
        :attr:`get_queue`, as long as the return value does not evaluate
        ``False``.
        """
        if self._CPU_avail >= event.CPU and self._memory_avail >= event.memory:
            # allocated requested resources
            self._CPU_avail -= event.CPU
            self._memory_avail -= event.memory

            # add the requster pod to the pods list
            self._pods.add(event.pod)

            # update node position in the nodepool heap
            if self._nodepool is not None:
                self._nodepool_heap_update()

            # succeed the request
            # event.succeed()
            event._ok = True
            event._value = None
            event.env.schedule(event, 0)  # schedule immeadiately

            # record the statistic
            self._mem_metric.Record(self._memory_capacity - self._memory_avail)
            self._cpu_metric.Record(self._CPU_capacity - self._CPU_avail)
            self._pod_metric.Record(self.pod_count)

            return True

        elif self._CPU_avail < event.CPU:
            # reject the request
            event._ok = False
            event._value = NodeCPUError('Node CPU Error!')
            event.env.schedule(event, 0)  # schedule immeadiately
            return None

        elif self._memory_avail < event.memory:
            # reject the request
            event._ok = False
            event._value = NodeOOMError('Node Memory Error!')
            event.env.schedule(event, 0)  # schedule immeadiately
            return None

    def _trigger_get(self, put_event):
        """Trigger get events.

        This method is called once a new get event has been created or a put
        event has been processed.

        The method iterates over all get events in the :attr:`get_queue` and
        calls :meth:`_do_get` to check if the conditions for the event are met.
        If :meth:`_do_get` returns ``False``, the iteration is stopped early.
        """
        idx = 0
        while idx < len(self.get_queue):
            get_event = self.get_queue[idx]
            proceed = self._do_get(get_event)
            if not get_event.triggered:
                idx += 1
            elif self.get_queue.pop(idx) != get_event:
                raise RuntimeError('Get queue invariant violated')

            if not proceed:
                break

    def record(self):
        # if node is running
        if self._shutdown_time == float('inf'):
            # record statistics
            self._mem_metric.Record(self._memory_capacity - self._memory_avail)
            self._cpu_metric.Record(self._CPU_capacity - self._CPU_avail)
            self._pod_metric.Record(self.pod_count)

            # TODO: what results should be reported?
            # calculate results
            UsageResult = namedtuple('UsageResult', ['average', 'max', 'min'])
            # memory
            avg_mem_usage = self._mem_metric.Mean()
            max_mem_usage = self._mem_metric.Max()
            min_mem_usage = self._mem_metric.Min()
            self._mem_results = UsageResult(avg_mem_usage,
                                            max_mem_usage,
                                            min_mem_usage)
            # cpu
            avg_cpu_usage = self._cpu_metric.Mean()
            max_cpu_usage = self._cpu_metric.Max()
            min_cpu_usage = self._cpu_metric.Min()
            self._cpu_results = UsageResult(avg_cpu_usage,
                                            max_cpu_usage,
                                            min_cpu_usage)

            # pod count
            avg_pod_count = self._pod_metric.Mean()
            max_pod_count = self._pod_metric.Max()
            min_pod_count = self._pod_metric.Min()
            self._pod_results = UsageResult(avg_pod_count,
                                            max_pod_count,
                                            min_pod_count)

    def clear(self):
        # memory
        self._mem_metric.Clear()

        # cpu
        self._cpu_metric.Clear()

        # pod count
        self._pod_metric.Clear()

        self.record()

    def get_results(self):
        '''
        returns a dictionary of results data
        '''
        # record the statistics upto current time
        self.record()

        NodeResult = namedtuple('NodeResults', ['memory', 'cpu', 'pod_count'])

        results = NodeResult(
            self._mem_results, self._cpu_results, self._pod_results)

        return {self.ID: results}

    def save_results(self, destination=None):
```

```python
        # record the node statistics
        self.record()

        if destination is None:
            destination = os.getcwd()

        # inside destination directory
        with change_dir(destination):
            node_dir = self._Node_ID
            # create a node directory
            os.mkdir(node_dir)

            # inside a node directory
            with change_dir(node_dir):

                # TODO: add avergae usgae
                # data (metrics)
                sim_time = self._mem_metric.T_list
                mem_usage = self._mem_metric.X_list
                mem_AUC = self._mem_metric.A_list
                cpu_usage = self._cpu_metric.X_list
                cpu_AUC = self._cpu_metric.A_list
                pod_count = self._pod_metric.X_list
                pod_AUC = self._pod_metric.A_list

                row_list = zip(sim_time, mem_usage, mem_AUC,
                               cpu_usage, cpu_AUC, pod_count, pod_AUC)

                target_file = self.ID + '-Metrics.csv'

                # create a csv file
                with open(target_file, 'w', newline='') as new_file:

                    fieldnames = ['sim_time', 'memory_usage', 'memory_AUC',
                                  'cpu_usage', 'cpu_AUC', 'pod_count', 'pod_AUC']

                    csv_writer = csv.writer(new_file)

                    csv_writer.writerow(fieldnames)
                    for row in row_list:
                        csv_writer.writerow(row)

                # create plots directory
                os.mkdir('plots')
                # inside plots directory
                with change_dir('plots'):
                    # save plots
                    self._plot_and_save(sim_time, mem_usage,
                                        'simulation time', 'memory usage')
                    self._plot_and_save(sim_time, cpu_usage,
                                        'simulation time', 'cpu usage')
                    self._plot_and_save(sim_time, pod_count,
                                        'simulation time', 'pod count')

    def _plot_and_save(self, x, y, xlabel, ylabel, destination=None):
        if destination is None:
            destination = os.getcwd()

        # new figure and associated axes
        fig1 = plt.figure(figsize=(12, 8))
        ax1 = fig1.add_subplot()

        # plot
        ax1.step(x, y, where='post')

        # set labels and titles
        ax1.set_xlabel(xlabel)
        ax1.set_ylabel(ylabel)
        ax1.set_title(f'total {ylabel} vs {xlabel} for {self.ID}')

        # save file at destination directory
        fig1.savefig(os.path.join(destination, ylabel + '.png'))

        # close the figure
        plt.close(fig1)
```

Nodepool.py

```
from heapq import heappush, heappop, nlargest, heapify
from itertools import count
from collections import namedtuple
import matplotlib.pyplot as plt
import csv
import os
import logging

# custom classes
from SimStatistics import CTStat, DTStat
```

```python
from Node import Node
from utils import (PoolMaxCapacityError,
                   PoolMinCapacityError,
                   NodePodCountNotZero,
                   NodeMatchNotFound,
                   change_dir)

# simpy classes
from simpy.core import BoundClass, Environment
from simpy.resources import base
from simpy.events import Event


# setup logger
logger = logging.getLogger(__name__)

logger.setLevel(logging.DEBUG)
formatter = logging.Formatter('%(levelname)s:%(message)s')

file_handler = logging.FileHandler('Simulation.log')
file_handler.setFormatter(formatter)

logger.addHandler(file_handler)

NodeSize = namedtuple('NodeSize', ['memory', 'cpu'])


class ScaleOut(Event):
    ''' Event for requesting to add a node to the pool
    '''

    def __init__(self, nodepool):
        super().__init__(nodepool._env)
        self.resource = nodepool
        self.proc = self.env.active_process

        nodepool.put_queue.append(self)
        nodepool._trigger_put(None)

    def __enter__(self):
        return self

    def __exit__(self,
                 exc_type,
                 exc_value,
                 traceback):
        self.cancel()
        return None

    def cancel(self):
        """Cancel this put request.

        This method has to be called if the put request must be aborted, for
        example if a process needs to handle an exception like an
        :class:`~simpy.exceptions.Interrupt`.

        If the put request was created in a :keyword:`with` statement, this
        method is called automatically.

        """
        if not self.triggered:
            self.resource.put_queue.remove(self)


class ScaleIn(Event):
    ''' Event for requesting to remove a node from the pool
    '''

    def __init__(self, nodepool):

        super().__init__(nodepool._env)
        self.resource = nodepool
        self.proc = self.env.active_process

        nodepool.get_queue.append(self)
        nodepool._trigger_get(None)

    def __enter__(self):
        return self

    def __exit__(
            self,
            exc_type,
            exc_value,
            traceback):
        self.cancel()
        return None

    def cancel(self):
        """Cancel this put request.
        This method has to be called if the put request must be aborted, for
        example if a process needs to handle an exception like an
        :class:`~simpy.exceptions.Interrupt`.

        If the put request was created in a :keyword:`with` statement, this
        method is called automatically.

        """
        if not self.triggered:
            self.resource.get_queue.remove(self)


class NodePool:
    ''' Pool containing nodes of same type between `min_count` and `max_count`.
    It supports the request to `scalein` and `scaleout` nodes from/into the pool.
    '''
    PutQueue = list
    GetQueue = list

    def __init__(self, env, name, node_size, node_cost, min_count, max_count, nod
e_init, start_delay=0):
        '''
        node_size is a NamedTuple(memory, cpu).
        node_init is a NamedTuple(memory, cpu) to consume from Node at all times.
 (overhead)
        '''
        self._env = env
        self._name = name

        # delay in start (ready for use) of a new node after bootup
        self._node_start_delay = start_delay

        # node type
        self._node_memory = node_size.memory
        self._node_cpu = node_size.cpu

        # node cost (per unit time)
        self._c = node_cost

        # node ovehead
        self._node_mem_init = node_init.memory
        self._node_cpu_init = node_init.cpu

        # auto scalling limits
        self._min_count = min_count
        self._max_count = max_count

        # list of active nodes in the pool
        self._nodes = []
        self._nid = count(1)  # for node id

        # list of all nodes (active and inactive)
        self._all_nodes = []

        # request queues
        self.put_queue = self.PutQueue()
        self.get_queue = self.GetQueue()

        # Bind event constructors as methods
        BoundClass.bind_early(self)

        # statistics
        # node count over time
        self._node_metric = CTStat(env, env.now, min_count)
        self._node_results = None
        # cost (avg per unit time for this pool)
        self._cost = None

        # add nodes to the pool
        for _ in range(min_count):
            self.scaleout()

    @property
    def name(self):
        return self._name

    @property
    def node_count(self):
        return len(self._nodes)

    @property
    def node_min_count(self):
        return self._min_count

    @property
    def node_max_count(self):
        return self._max_count

    @property
```

```python
    def node_size(self):
        return NodeSize(self._node_memory, self._node_cpu)

    @property
    def node_cost(self):
        return self._c

    @property
    def pool_cost(self):
        return self._cost

    @property
    def nodes(self):
        return self._nodes

    @property
    def pod_count(self):
        count = 0
        for node in self._nodes:
            count += node.pod_count
        return count

    @property
    def pods(self):
        pods = []
        for node in self._nodes:
            pods.extend(node.pods)
        return pods

    # put a node into the pool
    scaleout = BoundClass(ScaleOut)
    # get a node out of the node
    scalein = BoundClass(ScaleIn)

    def heap_update(self):
        '''
        update heap after any chnages to the node state.
        Called by node after state change.
        '''
        heapify(self._nodes)

    def get_most_utilized_node(self, demand=None):
        '''
        returns the node being most utilized
        scheduler call this method.
        'demand' is a NamedTuple(memory, cpu).
        '''
        # create a shallow copy of `self._nodes`
        nodes = self._nodes.copy()
        for _ in range(len(nodes)):
            candidate = heappop(nodes)
            if demand is None:
                return candidate
            elif (candidate.memory_avail > demand.memory) and (candidate.CPU_avai
l > demand.cpu):
                return candidate

        raise NodeMatchNotFound(
            f'None of the nodes on {self._name} can match the demand')

    def get_least_utilized_node(self, demand=None):
        '''
        returns the node being most utilized
        scheduler call this method.
        'demand' is a NamedTuple(memory, cpu).
        '''
        # get the node with the highest capacity
        candidate = nlargest(1, self._nodes)
        if demand is None:
            return candidate[0]
        elif (candidate[0].memory_avail > demand.memory) and (candidate[0].CPU_av
ail > demand.cpu):
            return candidate[0]

        raise NodeMatchNotFound(
            f'None of the nodes on {self._name} can match the demand')

    def _do_put(self, event):
        """Perform the *put* operation.

        This method is called by :meth:`_trigger_put` for every event in the
        :attr:`put_queue`, as long as the return value does not evaluate
        ``False``.
        """
        if self.node_count < self._max_count:
            # Instantiate a new node
            # TODO: change node name
            new_node = Node(self._env, self._name + '_Node_' + str(next(self._nid
)), self._node_cpu,
                            self._node_memory, self._node_cpu_init, self._node_me
m_init, self)

            # add a new node to the pool
            heappush(self._nodes, new_node)

            # update all_nodes list
            self._all_nodes.append(new_node)

            # succeed the request
            event._ok = True
            # TODO: check if works?
            event._value = new_node
            # schedule immeadiately
            # TODO: logic is incorrect for delay start
            event.env.schedule(event, priority=0, delay=self._node_start_delay)

            # record the statistic
            # TODO: add record for statistics
            self.record()

            return True

        else:
            # TODO: Handle the event manually
            event.fail(PoolMaxCapacityError(
                'Node count cannot exceed node_max_count!'))
            return None

    def _trigger_put(self, get_event):
        """This method is called once a new put(release) event has been created o
r a get(acquire)
        event has been processed.

        The method iterates over all put events in the :attr:`put_queue` and
        calls :meth:`_do_put` to check if the conditions for the event are met.
        If :meth:`_do_put` returns ``False``, the iteration is stopped early.
        """
        idx = 0
        while idx < len(self.put_queue):
            put_event = self.put_queue[idx]
            proceed = self._do_put(put_event)
            if not put_event.triggered:
                idx += 1
            elif self.put_queue.pop(idx) != put_event:
                raise RuntimeError('Put queue invariant violated')

            if not proceed:
                break

    def _do_get(self, event):
        """Perform the *get* operation.

        This method is called by :meth:`_trigger_get` for every event in the
        :attr:`get_queue`, as long as the return value does not evaluate
        ``False``.
        """
        if self.node_count > self._min_count:
            # most empty node (node with max available capacity)
            max_node = nlargest(1, self._nodes)[0]

            # check if pod count of a node with max capacity is 0
            if max_node.pod_count == 0:
                # record the node statistics
                max_node.record()
                # shutdown the node
                max_node.shutdown_time = self._env.now

                # remove node from the active nodes list
                self._nodes.remove(max_node)
                heapify(self._nodes)

                # DEBUG: print result and node count
                logger.debug(
                    f"Time= {self._env.now} | {max_node.ID} is removed (new node
count={self.node_count})")

                # succeed the request
                event._ok = True
                event._value = None
                # schedule immeadiately
                event.env.schedule(event, priority=0, delay=0)

                # record the pool statistic
                self.record()

                return True

            else:
                # don't raise runtime error
```

```python
                event._defused = True
                event._ok = False
                event._value = NodePodCountNotZero(
                    'All nodes have at least 1 pod running!')
                event.env.schedule(event, 0)  # schedule immeadiately
                return None

        else:
            # don't raise runtime error
            event._defused = True
            event._ok = False
            event._value = PoolMinCapacityError(
                'Node count cannot reduce below node_min_count!')
            event.env.schedule(event, 0)  # schedule immeadiately
            return None

    def _trigger_get(self, put_event):
        """Trigger get events.

        This method is called once a new get event has been created or a put
        event has been processed.

        The method iterates over all get events in the :attr:`get_queue` and
        calls :meth:`_do_get` to check if the conditions for the event are met.
        If :meth:`_do_get` returns ``False``, the iteration is stopped early.
        """
        idx = 0
        while idx < len(self.get_queue):
            get_event = self.get_queue[idx]
            proceed = self._do_get(get_event)
            if not get_event.triggered:
                idx += 1
            elif self.get_queue.pop(idx) != get_event:
                raise RuntimeError('Get queue invariant violated')

            if not proceed:
                break

    def record(self):
        # record the pool statistic
        self._node_metric.Record(self.node_count)

        # calculate results
        UsageResult = namedtuple('UsageResult', ['average', 'max', 'min'])
        # node count (upto current time)
        avg_node_count = self._node_metric.Mean()
        max_node_count = self._node_metric.Max()
        min_node_count = self._node_metric.Min()
        self._node_results = UsageResult(avg_node_count,
                                         max_node_count,
                                         min_node_count)

        # cost (per unit time)(averaged upto current time)
        self._cost = self._c*avg_node_count

    def clear(self):
        self._node_metric.Clear()

        self.record()

    def get_results(self):
        '''
        returns a dictionary of results data
        '''
        # record the statistics upto current time
        self.record()

        PoolResult = namedtuple('PoolResults', ['cost', 'node_count', 'nodes'])

        # get the metrics of each node inside the pool
        nodes_metrics = {}
        for node in self._all_nodes:
            nodes_metrics.update(node.get_results())

        # all pool level results
        results = PoolResult(self._cost, self._node_results, nodes_metrics)

        return {self._name: results}

    def save_results(self, destination=None):
        # record the statistics
        self._node_metric.Record(self.node_count)

        if destination is None:
            destination = os.getcwd()

        # inside destination directory
        with change_dir(destination):
            pool_dir = self._name
            # create pool directory
            os.mkdir(pool_dir)

            # inside pool directory
            with change_dir(pool_dir):

                # data (metrics)
                sim_time = self._node_metric.T_list
                node_count = self._node_metric.X_list
                node_AUC = self._node_metric.A_list
                avg_node_count = self._node_metric.t_avg_list

                row_list = zip(sim_time, node_count, node_AUC, avg_node_count)

                target_file = self.name + '-Metrics.csv'

                # create a csv file
                with open(target_file, 'w', newline='') as new_file:

                    fieldnames = ['sim_time', 'node_count',
                                  'node_AUC', 'avg_node_count']

                    csv_writer = csv.writer(new_file)

                    csv_writer.writerow(fieldnames)
                    for row in row_list:
                        csv_writer.writerow(row)

                # create plots directory
                os.mkdir('plots')

                # inside plots directory
                with change_dir('plots'):
                    # save plots
                    self._plot_and_save(sim_time, node_count,
                                        'simulation time', 'node count')
                    self._plot_and_save(sim_time, avg_node_count,
                                        'simulation time', 'average node count',
where='pre')

            # save node results
            for node in self._all_nodes:
                node.save_results(pool_dir)
```

```
    def _plot_and_save(self, x, y, xlabel, ylabel, destination=None, where='post'
):
        if destination is None:
            destination = os.getcwd()

        # new figure and associated axes
        fig1 = plt.figure(figsize=(12, 8))
        ax1 = fig1.add_subplot()

        # plot
        ax1.step(x, y, where=where)

        # set labels and titles
        ax1.set_xlabel(xlabel)
        ax1.set_ylabel(ylabel)
        ax1.set_title(f'{ylabel} vs {xlabel} for {self.name}')

        # save file at destination directory
        fig1.savefig(os.path.join(
            destination, xlabel + '_vs_' + ylabel + '.png'))

        # close the figure
        plt.close(fig1)
```

Cluster.py

```
import random
import os
import time
from collections import namedtuple
import logging

# custom classes
from utils import NodeMatchNotFound, change_dir


# setup logger
logger = logging.getLogger(__name__)

logger.setLevel(logging.DEBUG)
formatter = logging.Formatter('%(levelname)s:%(message)s')

file_handler = logging.FileHandler('Simulation.log')
file_handler.setFormatter(formatter)
```

```
logger.addHandler(file_handler)


class Cluster:
    '''
    Cluster is a group of nodepools.
    A nodepool contains nodes of same type and can scale up and down.
    '''

    def __init__(self, env, name, pools, seed=42):
        self._env = env
        self._name = name
        self._pools = pools

    @ property
    def name(self):
        return self._name

    @property
    def pools(self):
        '''
        returns the list of nodepools in the cluster
        '''
        return self._pools

    @property
    def cost(self):
        '''
        total cluster cost (per unit time)(averaged upto current time)
        '''
        cost = 0
        for pool in self._pools:
            cost += pool.pool_cost
        return cost

    @property
    def nodes(self):
        '''
        returns the list of active nodes in the cluster
        '''
        _nodes = []
        for pool in self._pools:
            _nodes.extend(pool.nodes)
        return _nodes
    @property
    def node_count(self):
        '''
        returns the count of total active nodes in the cluster
        '''
        count = 0
        for pool in self._pools:
            count += pool.node_count
        return count

    @property
    def pods(self):
        '''
        returns the list of running pods in the cluster
        '''
        pods = []
        for pool in self._pools:
            pods.extend(pool.pods)
        return pods

    @property
    def pod_count(self):
        '''
        returns the count of total running pods in the cluster
        '''
        count = 0
        for pool in self._pools:
            count += pool.pod_count
        return count

    def get_node(self, demand=None, criteria='most_utilized'):
        '''
        returns a node that match the demand based on criteria.

        '''
        # TODO: check demand type
        criteria_list = ['most_utilized', 'least_utilized',
                         'most_pods', 'least_pods', 'random']
        if criteria not in criteria_list:
            raise ValueError(
                'Invalid rule. Expected one of: %s' % criteria_list)

        if criteria == 'random':
            return random.choice(self.nodes)
        # sort based on memory_avail, cpu_avail, and bootup_time
        nodes = sorted(self.nodes)

        if criteria == 'most_utilized':
            if demand is None:
                return nodes[0]
            else:
                for candidate in nodes:
                    if (candidate.memory_avail > demand.memory) and (candidate.CP
U_avail > demand.cpu):
                        return candidate

            # no suitable node found
            # TODO: should I use exception here?
            raise NodeMatchNotFound(
                f'None of the nodes on {self._name} can match the demand')

        if criteria == 'least_utilized':
            if demand is None:
                return nodes[-1]
            else:
                for candidate in reversed(nodes):
                    if (candidate.memory_avail > demand.memory) and (candidate.CP
U_avail > demand.cpu):
                        return candidate

            # no suitable node found
            # TODO: should I use exception here?
            raise NodeMatchNotFound(
                f'None of the nodes on {self._name} can match the demand')

        # sort nodes based on pod count
        nodes = sorted(nodes, key=lambda node: node.pod_count)

        if criteria == 'most_pods':
            if demand is None:
                return nodes[-1]
            else:
                for candidate in reversed(nodes):
                    if (candidate.memory_avail > demand.memory) and (candidate.CP
U_avail > demand.cpu):
                        return candidate

            # no suitable node found
            # TODO: should I use exception here?
            raise NodeMatchNotFound(
                f'None of the nodes on {self._name} can match the demand')

        if criteria == 'least_pods':
            if demand is None:
                return nodes[0]
            else:
                for candidate in nodes:
                    if (candidate.memory_avail > demand.memory) and (candidate.CP
U_avail > demand.cpu):
                        return candidate

            # no suitable node found
            # TODO: should I use exception here?
            raise NodeMatchNotFound(
                f'None of the nodes on {self._name} can match the demand')

    def get_noodpool(self, criteria='most_pods'):
        '''
        returns a noodepool based on criteria
        '''
        criteria_list = ['most_pods', 'least_pods',
                         'max_size', 'min_size'
                         'most_nodes', 'least_nodes', 'random']

        if criteria not in criteria_list:
            raise ValueError(
                'Invalid rule. Expected one of: %s' % criteria_list)

        if criteria == 'random':
            return random.choice(self._pools)

        # sort nodepools based on node size
        pools = sorted(self._pools,
                       key=lambda pool: (pool.node_memory, pool.node_cpu))

        if criteria == 'min_size':
            return pools[0]

        if criteria == 'max_size':
            return pools[-1]

        # sort nodepools based on node count
        pools = sorted(self._pools,
                      key=lambda pool: pool.node_count)

        if criteria == 'least_nodes':
            return pools[0]

        if criteria == 'most_nodes':
            return pools[-1]

        # sort nodepools based on pod count
        pools = sorted(self._pools,
                       key=lambda pool: pool.pod_count)

        if criteria == 'least_pods':
            return pools[0]

        if criteria == 'most_pods':
            return pools[-1]

    def get_results(self):
        '''
        returns a dictionary of cluster level results
        '''
        ClusterResult = namedtuple('ClusterResults', ['cost', 'pools'])

        # get the metrics of each pool inside the cluster
        pools_metrics = {}
        for pool in self._pools:
            pools_metrics.update(pool.get_results())

        # all pool level results
        results = ClusterResult(self.cost, pools_metrics)

        return {self._name: results}

    def save_results(self, destination=None):
        if destination is None:
            destination = os.getcwd()

        # inside destination directory
        with change_dir(destination):
            cluster_dir = self._name
            # create a cluster directory
            os.mkdir(cluster_dir)

            # save pool results
            for pool in self._pools:
                pool.save_results(cluster_dir)
```

utils.py

```
from collections import namedtuple
from contextlib import contextmanager
import os
from simpy.exceptions import SimPyException

Resource = namedtuple('Resource', ['memory', 'cpu'])


class PoolMaxCapacityError(SimPyException):
    '''
    Pool cannot add more nodes than 'node_max_count'.
    '''

    def __init__(self, msg):
        super().__init__(msg)


class PoolMinCapacityError(SimPyException):
    '''
    Pool cannot remove nodes below 'node_min_count'.
    '''

    def __init__(self, msg):
        super().__init__(msg)


class NodePodCountNotZero(SimPyException):
    '''
    Pool cannot remove a node if pods are running on it.
    '''

    def __init__(self, msg):
        super().__init__(msg)


class NodeMatchNotFound(SimPyException):
    '''
    Pool cannot find a node that can fulfill the requested demand.
    '''
    def __init__(self, msg):
        super().__init__(msg)


class NodeOOMError(SimPyException):
    '''
    Node do not have enough memeory.
    '''

    def __init__(self, msg):
        super().__init__(msg)


class NodeCPUError(SimPyException):
    '''
    Node do not have enough CPU.
    '''

    def __init__(self, msg):
        super().__init__(msg)


@contextmanager
def change_dir(destination):
    try:
        cwd = os.getcwd()
        os.chdir(destination)
        yield
    finally:
        os.chdir(cwd)
```

SimStatistics.py

```
"""
Converted from VBASim Basic Classes
initially by Yujing Lin for Python 2.7
Update to Python 3 by Linda Pei & Barry L Nelson

Modified by Vijaykumar Maraviya for use with Simpy
Last update 3/25/2021
"""
import math
```

```
class CTStat():
    # Generic continuous-time statistics class
    # Note that CTStat should be called AFTER the value of the variable changes

    def __init__(self, env, Tinit, Xinit):
        # Excecuted when the CTStat object is created to initialize variables
        self.Area = 0.0
        self.Tlast = Tinit  # usually entity start time
        self.TClear = Tinit
        self.Xlast = Xinit
        self.env = env  # env.now returns the current simulation time

        # for plotting
        self.T_list = [Tinit]
        self.X_list = [Xinit]
        self.A_list = [0]
        self.t_avg_list = [0]

    def Record(self, X):
        # Update the CTStat from the last time change and keep track of previous
value
        self.Area = self.Area + self.Xlast * (self.env.now - self.Tlast)
        self.Tlast = self.env.now
        self.Xlast = X

        if self.Tlast == self.T_list[-1]:
            # update X in place
            self.X_list[-1] = self.Xlast
        else:
            # add new entry to the lists
            self.T_list.append(self.Tlast)
            self.X_list.append(self.Xlast)
            self.A_list.append(self.Area)
            self.t_avg_list.append(self.Area/self.Tlast)

    def Mean(self):
        # Return the sample mean up through the current time but do not update
        mean = 0.0
        if (self.env.now - self.TClear) > 0.0:
            mean = (self.Area + self.Xlast * (self.env.now -
                                              self.Tlast)) / (self.env.now - self
.TClear)
        return mean

    def Max(self):
```

```python
        max_val = 0.0
        if (self.env.now - self.TClear) > 0.0:
            max_val = max(self.X_list)
        return max_val

    def Min(self):
        min_val = 0.0
        if (self.env.now - self.TClear) > 0.0:
            min_val = min(self.X_list)
        return min_val

    def Clear(self):
        # Clear statistics during the simulation
        self.Area = 0.0
        self.Tlast = self.env.now
        self.TClear = self.env.now

        self.T_list = []
        self.X_list = []
        self.A_list = []
        self.t_avg_list = []


class DTStat():
    # Generic discrete-time statistics class

    def __init__(self, env, Tinit):
        # Excecutes when the DTStat object is created to initialize variables
        self.Sum = 0.0
        self.SumSquared = 0.0
        self.NumberOfObservations = 0.0

        # to keep track of time (not use in computation)
        self.Tlast = Tinit
        self.env = env  # env.now returns the current simulation time

        # timestamp of Record method calls
        self.T_list = []
        # store indicator values
        self.X_list = []
        # store running average
        self.RunningAvg = []

    def Record(self, X):
        # Update the DTStat
        self.Sum = self.Sum + X
        self.SumSquared = self.SumSquared + X * X
        self.NumberOfObservations = self.NumberOfObservations + 1

        # time of `Record` call
        self.Tlast = self.env.now

        self.X_list.append(X)
        self.T_list.append(self.Tlast)
        self.RunningAvg.append(self.Mean())

    def Mean(self):
        # Return the sample mean
        mean = 0.0
        if self.NumberOfObservations > 0.0:
            mean = self.Sum / self.NumberOfObservations
        return mean

    def StdDev(self):
        # Return the sample standard deviation
        stddev = 0.0
        if self.NumberOfObservations > 1.0:
            stddev = math.sqrt((self.SumSquared - self.Sum**2 /
                                    self.NumberOfObservations) / (self.NumberOfObserv
ations - 1))
        return stddev

    def N(self):
        # Return the number of observations collected
        return self.NumberOfObservations

    def Clear(self):
        # Clear statistics
        self.Sum = 0.0
        self.SumSquared = 0.0
        self.NumberOfObservations = 0.0

        self.Tlast = self.env.now
        # timestamp of Record method calls
        self.T_list = []
        # store indicator values
        self.X_list = []
        # store running average
        self.RunningAvg = []
```

[Analysis] Select data deletion.py

```python
# %%
import random
import itertools
import os
import time
import numpy as np
import logging

# custom classes
from Node import Node
from NodePool import NodePool
from Cluster import Cluster
from Schedular import BestFitBinPackingSchedular
from PodGenerator import PodGenerator
from utils import Resource, change_dir
import SimRNG

# simpy classes
import simpy.core as core

logger = logging.getLogger(__name__)
logger.setLevel(logging.DEBUG)
formatter = logging.Formatter('%(levelname)s:%(message)s')
file_handler = logging.FileHandler('simulation.log')
file_handler.setFormatter(formatter)
logger.addHandler(file_handler)

# %%


def ArrivalGen():
    '''
    generate and yield interarrival time for pods

    '''
    # average time between two consective pod submission
    MeanTBA = 3/100

    while True:
        yield SimRNG.Expon(MeanTBA, Stream=10)
```

```
def PeriodGen():
    '''
    generate and yield pod life (activity period)
    Erlang distribution
    '''
    # User spend (pod life is) on average 3 hours
    MeanAP = 3
    Phases = 3

    while True:
        yield SimRNG.Erlang(Phases, MeanAP, Stream=20)


rg1 = np.random.RandomState()
rg2 = np.random.RandomState()


def DemandGen():
    '''
    generate and yield pod resource demand
    '''
    while True:
        mem = rg1.uniform(512, 2048)
        cpu = rg2.uniform(0.05, 0.30)
        yield Resource(mem, cpu)


period_gen = PeriodGen()
resource_gen = DemandGen()
arrival_gen = ArrivalGen()

# %%

# node types (for cluster configuration)
node_types = {'E2s_v4': {'cost': 0.1767,
                         'size': Resource(16384, 2),
                         'reserved': Resource(3262.40, 0.20)},
              'E4s_v4': {'cost': 0.3533,
                         'size': Resource(32768, 4),
                         'reserved': Resource(4245.44, 0.24)},
              'E8s_v4': {'cost': 0.7066,
                         'size': Resource(65536, 8),
                         'reserved': Resource(6211.52, 0.28)},
              'E16s_v4': {'cost': 1.4132,
                          'size': Resource(131072, 16),
                          'reserved': Resource(10143.68, 0.36)},
              'E20s_v4': {'cost': 1.7664,
                          'size': Resource(163840, 20),
                          'reserved': Resource(10801.60, 0.42)},
              'E32s_v4': {'cost': 2.8263,
                          'size': Resource(262144, 32),
                          'reserved': Resource(12765.12, 0.52)},
              'E48s_v4': {'cost': 4.2394,
                          'size': Resource(393216, 48),
                          'reserved': Resource(15386.56, 0.67)},
              'E64s_v4': {'cost': 5.6525,
                          'size': Resource(516096, 64),
                          'reserved': Resource(17844.16, 0.84)}
              }

# %%

# for all scenarios run simulation
for type, config in node_types.items():

    # use common random number across scenarios
    SimRNG.ZRNG = SimRNG.InitializeRNSeed()
    rg1.seed(1)
    rg2.seed(2)

    env = core.Environment()

    # config for node pool
    pool_A = NodePool(env, type,
                      node_size=config['size'], node_cost=config['cost'],
                      min_count=1, max_count=100, node_init=config['reserved'])

    # list of pools in cluster
    pools = [pool_A]

    # cluster
    cluster = Cluster(env, 'cluster' + type, pools)

    # schedular
    schedular = BestFitBinPackingSchedular(env, 'schedular', cluster)

    # pod attributes (limit and guarantee)
    resource_guarantee = Resource(512, 0.05)
    resource_limit = Resource(2048, 0.30)
    # pod generator
    pod_generator = PodGenerator(env, 'Static Pod Generator', arrival_gen)
    env.process(pod_generator.generate(resource_guarantee,
                                       resource_limit,
                                       period_gen,
                                       resource_gen,
                                       schedular
                                       )
                )

    # status update on consol
    print(f'running simulation for config type: {type}')

    # run simulation
    env.run(until=100)

    # save results
    print('saving results ...')
    dest_path = os.getcwd()
    results_path = os.path.join(dest_path, 'Simulation Results')

    if not os.path.exists(results_path):
        os.makedirs(results_path)

    with change_dir(results_path):
        timestr = time.strftime("%Y%m%d-%H%M%S")
        sim_dir = 'simulation_' + type + '_' + timestr
        os.mkdir(sim_dir)

        pod_generator.save_results(sim_dir)
        cluster.save_results(sim_dir)

    print(f'results for config type {type} are saved at {results_path}')

    # print results
    res = cluster.get_results()

    # print cost
    print(f"Cost for config type {type}={res['cluster' + type].cost}")

# %%
```

[Analysis] cleanup.py

```
# %%
```

```python
import random
import itertools
import os
import time
import numpy as np
import scipy.stats as sp
import logging

# custom classes
from Node import Node
from NodePool import NodePool
from Cluster import Cluster
from Schedular import BestFitBinPackingSchedular
from PodGenerator import PodGenerator
from utils import Resource, change_dir
import SimRNG

# simpy classes
import simpy.core as core

logger = logging.getLogger(__name__)
logger.setLevel(logging.DEBUG)
formatter = logging.Formatter('%(levelname)s:%(message)s')
file_handler = logging.FileHandler('simulation.log')
file_handler.setFormatter(formatter)
logger.addHandler(file_handler)

# %%


def mean_confidence_interval(data, confidence=0.95):
    a = 1.0*np.array(data)
    n = len(a)
    m, se = np.mean(a), sp.sem(a)
    h = se * sp.t._ppf((1+confidence)/2., n-1)
    return m, m-h, m+h


def ArrivalGen():
    '''
    generate and yield interarrival time for pods

    '''
    # average time between two consective pod submission
    MeanTBA = 3/100
    while True:
        yield SimRNG.Expon(MeanTBA, Stream=10)


def PeriodGen():
    '''
    generate and yield pod life (activity period)
    Erlang distribution
    '''
    # User spend (pod life is) on average 3 hours
    MeanAP = 3
    Phases = 3

    while True:
        yield SimRNG.Erlang(Phases, MeanAP, Stream=20)


rg1 = np.random.RandomState()
rg2 = np.random.RandomState()


def DemandGen():
    '''
    generate and yield pod resource demand
    '''
    while True:
        mem = rg1.uniform(512, 2048)
        cpu = rg2.uniform(0.05, 0.30)
        yield Resource(mem, cpu)


period_gen = PeriodGen()
resource_gen = DemandGen()
arrival_gen = ArrivalGen()

# %%

# node types (for cluster configuration)
node_types = {'E16s_v4': {'cost': 1.4132,
                          'size': Resource(131072, 16),
                          'reserved': Resource(10143.68, 0.36)},
              'E20s_v4': {'cost': 1.7664,
                          'size': Resource(163840, 20),
                          'reserved': Resource(10801.60, 0.42)}}
```

```python
# %%

# for all scenarios run simulation
for type, config in node_types.items():

    all_success_prob = []
    all_cluster_cost = []

    # use common random number across scenarios
    SimRNG.ZRNG = SimRNG.InitializeRNSeed()
    rg1.seed(1)
    rg2.seed(2)

    # run 10 times
    for i in range(10):

        env = core.Environment()

        # config for node pool
        pool_A = NodePool(env, type,
                          node_size=config['size'], node_cost=config['cost'],
                          min_count=1, max_count=100, node_init=config['reserved'
])

        # list of pools in cluster
        pools = [pool_A]

        # cluster
        cluster = Cluster(env, 'cluster' + type, pools)

        # schedular
        schedular = BestFitBinPackingSchedular(env, 'schedular', cluster)

        # pod attributes (limit and guarantee)
        resource_guarantee = Resource(512, 0.05)
        resource_limit = Resource(2048, 0.30)
        # pod generator
        pod_generator = PodGenerator(env, 'Static Pod Generator', arrival_gen)
        env.process(pod_generator.generate(resource_guarantee,
                                           resource_limit,
                                           period_gen,
                                           resource_gen,
                                           schedular
                                           )
                    )
        # status update on consol
        print(f'running simulation for config type: {type}')

        # run simulation
        env.run(until=500)

        # get results
        cluster_res = cluster.get_results()
        pod_gen_res = pod_generator.get_results()

        # avg cluster cost
        avg_cost = cluster_res['cluster' + type].cost
        # success probabaility
        success_prob = pod_gen_res['Static Pod Generator'].success_probability

        all_cluster_cost.append(avg_cost)
        all_success_prob.append(success_prob)

        # # print
        # print(f"run: {i}, Cost for config type {type}={avg_cost:.3f}")
        # print(
        #     f"run: {i}, success probability for config type {type}={success_pro
b:.3f}")

    m, lb, ub = mean_confidence_interval(all_cluster_cost)
    print(
        f'estimated expected unit cost for config type{type} = {m:.3f} $ and 95%
CI of estimate is [{lb:.3f}, {ub:.3f}]')

    m, lb, ub = mean_confidence_interval(all_success_prob)
    print(
        f'estimated expected success probability for config type{type} = {m:.3f}
and 95% CI of estimate is [{lb:.3f}, {ub:.3f}]')


# %%
```

[Analysis] Multipool.py

```
# %%
import random
import itertools
import os
```

```python
import time
import numpy as np
import scipy.stats as sp

# custom classes
from Node import Node
from NodePool import NodePool
from Cluster import Cluster
from Schedular import BestFitBinPackingSchedular, CustomSchedular
from PodGenerator import PodGenerator
from utils import Resource, change_dir
import SimRNG

# simpy classes
import simpy.core as core

# %%

def mean_confidence_interval(data, confidence=0.95):
    a = 1.0*np.array(data)
    n = len(a)
    m, se = np.mean(a), sp.sem(a)
    h = se * sp.t._ppf((1+confidence)/2., n-1)
    return m, m-h, m+h

def ArrivalGen():
    '''
    generate and yield interarrival time for pods

    '''
    # average time between two consective pod submission
    MeanTBA = 3/100

    while True:
        yield SimRNG.Expon(MeanTBA, Stream=10)


def PeriodGen():
    '''
    generate and yield pod life (activity period)
    Erlang distribution
    '''
    # User spend (pod life is) on average 3 hours
    MeanAP = 3
    Phases = 3
    while True:
        yield SimRNG.Erlang(Phases, MeanAP, Stream=20)


rg1 = np.random.RandomState()
rg2 = np.random.RandomState()


def DemandGen():
    '''
    generate and yield pod resource demand
    '''
    while True:
        mem = rg1.uniform(512, 2048)
        cpu = rg2.uniform(0.05, 0.30)
        yield Resource(mem, cpu)


period_gen = PeriodGen()
resource_gen = DemandGen()
arrival_gen = ArrivalGen()

# %%

# node types (for cluster configuration)
node_types = {'E2s_v4': {'cost': 0.1767,
                         'size': Resource(16384, 2),
                         'reserved': Resource(3262.40, 0.20)},
              'E4s_v4': {'cost': 0.3533,
                         'size': Resource(32768, 4),
                         'reserved': Resource(4245.44, 0.24)},
              'E8s_v4': {'cost': 0.7066,
                         'size': Resource(65536, 8),
                         'reserved': Resource(6211.52, 0.28)},
              'E16s_v4': {'cost': 1.4132,
                          'size': Resource(131072, 16),
                          'reserved': Resource(10143.68, 0.36)},
              'E20s_v4': {'cost': 1.7664,
                          'size': Resource(163840, 20),
                          'reserved': Resource(10801.60, 0.42)},
              'E32s_v4': {'cost': 2.8263,
                          'size': Resource(262144, 32),
                          'reserved': Resource(12765.12, 0.52)},
              'E48s_v4': {'cost': 4.2394,
                          'size': Resource(393216, 48),
                          'reserved': Resource(15386.56, 0.67)},
              'E64s_v4': {'cost': 5.6525,
                          'size': Resource(516096, 64),
                          'reserved': Resource(17844.16, 0.84)}
              }

# %%
type = 'E2s_E20s_v4'
config1 = node_types['E20s_v4']
config2 = node_types['E2s_v4']


# use common random number across scenarios
SimRNG.ZRNG = SimRNG.InitializeRNSeed()
rg1.seed(1)
rg2.seed(2)

all_success_prob = []
all_cluster_cost = []

# run 10 times
for i in range(10):

    env = core.Environment()

    # config for node pool
    pool_A = NodePool(env, 'E20s_v4',
                      node_size=config1['size'], node_cost=config1['cost'],
                      min_count=1, max_count=100, node_init=config1['reserved'])

    # config for node pool
    pool_B = NodePool(env, 'E2s_v4',
                      node_size=config2['size'], node_cost=config2['cost'],
                      min_count=0, max_count=2, node_init=config2['reserved'])

    # list of pools in cluster
    pools = [pool_A, pool_B]

    # cluster
    cluster = Cluster(env, 'cluster' + type, pools)

    # schedular
    schedular = CustomSchedular(env, 'schedular', cluster)
```

```python
    # pod attributes (limit and guarantee)
    resource_guarantee = Resource(512, 0.05)
    resource_limit = Resource(2048, 0.30)
    # pod generator
    pod_generator = PodGenerator(env, 'Static Pod Generator', arrival_gen)
    env.process(pod_generator.generate(resource_guarantee,
                                       resource_limit,
                                       period_gen,
                                       resource_gen,
                                       schedular
                                       )
                )

    # status update on consol
    print(f'running simulation for config type: {type}')

    # run simulation
    env.run(until=100)

    # get results
    cluster_res = cluster.get_results()
    pod_gen_res = pod_generator.get_results()

    # avg cluster cost
    avg_cost = cluster_res['cluster' + type].cost
    # success probabaility
    success_prob = pod_gen_res['Static Pod Generator'].success_probability

    all_cluster_cost.append(avg_cost)
    all_success_prob.append(success_prob)

m, lb, ub = mean_confidence_interval(all_cluster_cost)
print(
    f'estimated expected unit cost for config type{type} = {m:.3f} $ and 95% CI of estimate is [{lb:.3f}, {ub:.3f}]')

m, lb, ub = mean_confidence_interval(all_success_prob)
print(
    f'estimated expected success probability for config type{type} = {m:.3f} and 95% CI of estimate is [{lb:.3f}, {ub:.3f}]')
```