Project Submission Form — Hospital Queue Simulation

> Filled form (English). Copy this Markdown content directly into your submission files or export as needed.

Project title

Simulation of Hospital Patient Flow — M/M/c Queue Network

---

1. Team work declaration

Team

Student	Id
SV1	ID1
SV2	ID2
SV3	ID3
SV4	ID4
SV5	ID5
SV6	ID6

(Replace SVx / IDx with actual student names and IDs.)

Workmap declaration

Last name	Student ID	Design Role	Dev Role	Eval Role	Report Role	Ass1 work description (details)	Rating	week 2 outcome	week 3 outcome
S1	ID1	yes	yes	yes	yes	System architecture, `sim_engine.py`, final report integration	A	Repo + config skeleton	sim run & debug
S2	ID2	yes	yes	no	yes	Implement `queue_node.py`, `patient.py`, event integrals, unit tests	A	QueueNode design	Implement serve/ process & tests

Last name	Student ID	Design Role	Dev Role	Eval Role	Report Role	Ass1 work description (details)	Rating	week 2 outcome	week 3 outcome
S3	ID3	no	yes	yes	no	Implement `arrival.py`, `router.py`, `metrics.py`, CSV export & aggregation	B	Arrival generator	Metrics & CSV export
S4	ID4	no	yes	no	no	Test harness, CI-like test scripts, fix bugs	B	Create tests	Run tests & fix issues
S5	ID5	yes	no	yes	yes	Analytic M/M/c formulas, compare sim vs theory, write theoretical sections	A	Analytical formulas	Compute benchmarks
S6	ID6	no	no	yes	no	Post-processing, notebook plots, CI computation & summary tables	B	Notebook skeleton	Plotting & CI

2. System design

System Performance Steps

1. Define Goal and System
Model a simplified hospital patient flow as four queueing nodes: Registration → Doctor → (Lab with probability p_lab) → Pharmacy. Goal: measure waiting times, queue lengths, utilizations and identify bottlenecks under different workloads.

2. Service and Outcomes
- Services: Registration (M/M/3), Doctor (M/M/5), Lab (M/M/4), Pharmacy (M/M/2).
- Outcomes: per-node mean waiting time Wq, service time S, response time R, time-average queue length Lq, server utilization u, throughput.

3. Select Metrics
- Per-patient: waiting per node, service per node, total response R.
- Per-node (time-average): Lq (∫ q(t)dt / T), average busy servers (∫ s(t)dt / T), utilization = avg_busy / servers.
- Overall: E[w], E[R], throughput, CI across replications.

4. List System Parameters
- External: arrival_rate λ, p_lab.
- Node: servers c_i, service_rate μ_i.
- Simulation: warmup_time, run_time, replications, seed.

5. List Factors to Study
- Vary λ (balanced → stress).
- Vary p_lab.
- Vary c_i or μ_i for capacity planning.

6. Evaluation Technique
- Simulate with SimPy; run multiple replications; remove warmup; compute time-average integrals; compare with closed-form M/M/c benchmarks.

7. Select Workload
- Workload A (balanced): λ selected to keep ρ < 0.7 in nodes.
- Workload B (stress): increase λ until a node hits ρ ~ 1.

System Component / Module description

- `sim_engine.py` — Experiment runner, env setup, replications, CLI.
- `config.py` — Default config (λ, nodes, μ, c, p_lab, run/warmup defaults).
- `patient.py` — Patient data holder for timestamps.
- `queue_node.py` — M/M/c wrapper with event-driven integrals.
- `arrival.py` — Homogeneous Poisson arrival generator.
- `router.py` — Routing logic with p_lab.
- `metrics.py` — Collects metrics, writes CSV, computes summary & CI.
- `experiments.py` — Scripts for workload sweeps.
- `notebooks/` — post-processing / plotting (optional).
- `tests/` — unit tests.

---

3. System Implementation

> The instructor provided empty boxes to paste code/commands. Below are ready-to-copy code snippets and CLI commands. Save them as files under `src/`.

Install prerequisite libraries

Option A — install individual packages

```
# create and activate virtual environment (macOS / Linux)
python3 -m venv .venv
source .venv/bin/activate

# install core packages
pip install simpy pandas

# optional / recommended for analysis and tests
pip install numpy matplotlib pytest
```

Option B — install from requirements.txt

Create a `requirements.txt` with the following content:

```
simpy>=4.1
pandas>=1.3
numpy>=1.22
matplotlib>=3.0
pytest
```

Then install all with:

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

Notes / troubleshooting - On macOS, some NumPy wheels may not match the system Python (e.g., Python 3.14). If you see ImportError from NumPy C-extensions, recreate the venv with Python 3.12 or install NumPy from source (requires build tools). Recommended: use Python 3.12. - If you do not need NumPy for running the simulation, the code uses `random.expovariate()` as fallback.

Queuing Node 1 — Registration (file: `src/queue_node.py`)

Create `src/queue_node.py` and paste the class below:

```
import simpy
import random

class QueueNode:
    def __init__(self, env, name, service_rate, servers):
        self.env = env
        self.name = name
        self.service_rate = float(service_rate)
        self.servers = int(servers)
        self.resource = simpy.Resource(env, capacity=self.servers)
        # event-driven integrals
        self.last_event_time = env.now
        self.queue_area = 0.0
        self.busy_area = 0.0
        self.current_in_service = 0
        self.completed_jobs = 0

    def _update_areas(self):
        now = self.env.now
        dt = now - self.last_event_time
        if dt > 0:
            q_len = len(self.resource.queue)
            self.queue_area += q_len * dt
            self.busy_area += self.current_in_service * dt
            self.last_event_time = now

    def _sample_service_time(self):
        if self.service_rate <= 0:
            return 0.0
```

```
        return random.expovariate(self.service_rate)

    def serve(self, patient):
        patient.record_arrival(self.name, self.env.now)
        self._update_areas()
        with self.resource.request() as req:
            yield req
            self._update_areas()
            patient.record_service_start(self.name, self.env.now)
            self.current_in_service += 1
            self._update_areas()
            st = self._sample_service_time()
            yield self.env.timeout(st)
            patient.record_service_end(self.name, self.env.now)
            self.current_in_service -= 1
            self.completed_jobs += 1
            self._update_areas()

    def finalize(self, sim_end_time):
        if sim_end_time > self.last_event_time:
            dt = sim_end_time - self.last_event_time
            self.queue_area += len(self.resource.queue) * dt
            self.busy_area += self.current_in_service * dt
            self.last_event_time = sim_end_time
```

Use this same class to instantiate registration, doctor, lab, and pharmacy nodes with appropriate `service_rate` and `servers`.

Queuing Node 2 — Doctor

Instantiate with:

```
# in sim_engine setup
doctor = QueueNode(env, 'doctor', service_rate=5.0, servers=5)
```

Chain doctor after registration by calling `yield env.process(doctor.serve(patient))` inside the patient flow after registration.

Queuing Node 3 — Lab (router)

Create `src/router.py` with:

```
import random

def route_after_doctor(env, patient, lab_node, pharmacy_node, p_lab=0.2):
    if random.random() < p_lab:
        yield env.process(lab_node.serve(patient))
    yield env.process(pharmacy_node.serve(patient))
```

Queuing Node 4 — Pharmacy

Instantiate with:

```
pharmacy = QueueNode(env, 'pharmacy', service_rate=6.0, servers=2)
```

Take `exit_time` as `patient.timestamps['pharmacy_service_end']`.

System configuration and execution (sim_engine usage)

Create `src/sim_engine.py` with an orchestrator that: - Builds `simpy.Environment` per replication - Instantiates nodes using values from `config.py` - Spawns an `arrival_generator` that creates `Patient` objects and starts registration process - Chains doctor, router, lab, pharmacy in patient flow - Runs env until `warmup_time + run_time` - Calls `finalize(sim_end_time)` on nodes and writes CSVs via `metrics.py`

Run from CLI:

```
python src/sim_engine.py --run_time 2000 --warmup_time 200 --replications 10
--seed 100 --output outputs/results_csv --workload demo
```

Programmatic API example:

```
from sim_engine import run_experiment
out = run_experiment(run_time=2000.0, warmup_time=200.0, replications=10,
base_seed=100, output_dir='outputs/results_csv', workload_name='demo')
print(out['summary_file'])
```

---

4. Present results

Below are the code snippets and commands for producing outputs and basic plots.

Run experiments (CLI)

Baseline:

```
python src/sim_engine.py --run_time 2000 --warmup_time 200 --replications 10
--seed 100 --output outputs/results_csv/baseline --workload baseline
```

Stress sweep:

```
# simple loop in shell
for lam in 6 8 10 12; do
```

```
  python -c "from config import config; config['arrival_rate']=$lam; import
sim_engine; sim_engine.run_experiment(2000,200,5,200,'outputs/results_csv/
lam_$lam','lam_$lam')"
done
```

Or use experiments.py (create `src/experiments.py` as below):

```
# src/experiments.py
from config import config
from sim_engine import run_experiment
for lam in [4.0, 6.0, 8.0, 10.0]:
    config['arrival_rate'] = lam
    out = run_experiment(2000, 200, 5, 1000+int(lam*10), f'outputs/
results_csv/lam_{lam}', workload_name=f'lam_{lam}')
    print('Saved', out['summary_file'])
```

Postprocessing & plotting (example)

Create `scripts/postprocess.py`:

```
import pandas as pd
from pathlib import Path
base = Path('outputs/results_csv')
# aggregate per-node rep files
dfs = []
for f in (base / 'per_node_rep').glob('*.csv'):
    dfs.append(pd.read_csv(f))
all_nodes = pd.concat(dfs, ignore_index=True)
summary = all_nodes.groupby('node_name')
['mean_waiting_time'].agg(['mean','std','count'])
print(summary)
```

Create `scripts/plot_wait_times.py`:

```
import pandas as pd
import matplotlib.pyplot as plt
node_df = pd.read_csv('outputs/results_csv/per_node_rep/
demo_rep0_seed100.csv')
plt.bar(node_df['node_name'], node_df['mean_waiting_time'])
plt.ylabel('Mean waiting time (time units)')
plt.xlabel('Node')
plt.title('Mean waiting time per node (rep0)')
plt.show()
```

---

5. Conclusions

Coordinator statement
The team implemented a reproducible modular SimPy simulation of a 4-node hospital queue network that records per-patient timestamps and computes accurate time-average metrics. Baseline experiments show low waiting times; stress tests identify bottlenecks when arrival rate grows.

Strengths: 1. Event-driven integrals for time-average metrics (accurate Lq and utilization).
2. Reproducible via seeds and replications; CSV outputs for detailed analysis.
3. Modular code structure.

Limitations: 1. Homogeneous Poisson arrivals only in baseline.
2. No server schedule/shift modeling or priority queues.
3. Per-patient logging may not scale to very large simulations without sampling.

If you want, I can export this Markdown as a downloadable file (project_form.md) or generate a PDF / Word doc for submission. Also I can replace placeholder student names/IDs with your team list if you paste them.