

Simulation and Modelling: Final Report

Federico Giacardi

University of Turin, academic year 2024-2025

Abstract. What follows summarizes the implementation and validation of a discrete-events simulator of a flexible manufacturing system. The simulator's application to evaluate the system's performance is also briefly discussed.

1 Basic Features and Events

1.1 Basic Features

The program implements a next-event simulator of a system with four single server queue stations and two infinite server stations. The simulator is written in C++, and organized into four different modules:

- **NESsq_List_Manager** which provides the implementation of basic list management operations
- **NESsq** which implements the core of the simulator
- **random_var_gen** that contains the code for the generation of Negative Exponential, Hyper-exponential, and Uniform random variables, as explained in class
- **rngs** which is the random number generator by Park and Miller, as provided during the course

The project is structured into two different directories:

- **Simulator** which contains the four modules listed above and can be compiled and executed using:

```
make run PRINTOUT=[0/1]
```

where the boolean flag can be passed as an argument to obtain a printout of the quantities collected during each regeneration cycle (the flag is set to false by default).

- **MVA** which contains the implementation of the MVA algorithm. The computation of visit counts using Gauss elimination was omitted due to a bug. These values were therefore hard-coded into the program. It can be compiled and launched by executing

```
make run
```

The structure of the core was built out of the skeleton provided as a base for the exercises of the course, with some modifications described in the following. All the parameters of the model are represented with a macro. Each event is represented by an event notice instance of this struct

```
typedef struct {
    int type;
    char name[256];
    double create_time;
    double occur_time;
    double arrival_time;
    double service_time;
    double rem_service_time;
    double start_man;
    double arrival_load;
    int num_visits_M1;
    bool isTagged;
} event_notice;
```

The tagged customer approach was used to collect the average manufacturing times needed to build the required estimator. The sample of observations collected during all the regeneration cycles is represented by an instance of the following data structure

```
typedef struct {
    int num_cycles;
    double S_a;
    double S_v;
    double S_aa;
    double S_av;
    double S_vv;
    double width;
    double mi;
    double stdev;
    double precision;
    std::list<regeneration_pair> observations;
} regeneration_sample;

with

typedef struct {
    double man_time_sum;
    int passages_counter;
} regeneration_pair;
```

with the semantic of the fields introduced during Lecture 22. All the data needed to compute the performance measures of each station are collected using the following data structure

```
typedef struct {
    int Area_n;
    int Area_q;
    int Area_y;
    int nsys;
    int narr;
    int ncomp;
    performance_sum station_sum;
} SSQ_station;
```

with Area_n being the area under the function that counts the total number of jobs currently in the station, Area_q that of the function that counts the number of jobs waiting in the station queue, and Area_y that of the function that counts the number of jobs currently in service. Lastly, station_sum has the following structure

```
typedef struct {
    double delay;
    double wait;
    double service;
} performance_sum;
```

Given that the load and unload station are delay stations, their sum of service times will always be zero. The global state of the system has been adapted with the introduction of the following variables:

- **remaining_lifetime**, which is the remaining operative time of the tool of machine M1 before its substitution
- **nPartsStarted** which counts the number of parts whose manufacturing process has been started
- **nPartsCompleted** which counts the number of parts whose manufacturing process has been completed

The initialization procedure has been extended to schedule the immediate arrival of all the AGVs active in the system at the loading station. As requested by the process specification, each random variable instance is generated starting from a random number extracted from a different stream, set using the Select-Stream method. The report function prints the performance statistics of all the stations, as well as the average manufacturing, response, and cycle times of the whole system, computed both as *total_sum/number_of_elements* and as average of waiting times of the stations of interest. In particular, for the average manufacturing time, the following formulas have been used

$$\frac{total_manufacturing_time}{nPartsCompleted} \quad (1)$$

$$\frac{M1.station_sum.wait}{M1.ncomp} + \frac{M2.station_sum.wait}{M2.ncomp} + \frac{M3.station_sum.wait}{M3.ncomp} \quad (2)$$

with the second one derived by observing that the total manufacturing time is the total amount of time spent by a part in the subsystem made up by stations M1, M2, and M3. The report function also prints bounds, mean, and precision of confidence interval for the average manufacturing time computed with the regeneration method, which has been implemented according to the structure suggested in Homework 7, and using the formulas of Lecture 22.

1.2 Events List


We start the description of our system by listing all the events that dictate the flow of the simulation, along with their effect on the system state and performance statistics. Each of them is represented by an integer, encoded with a macro.

arrival_load Arrival of an empty AGV to the loading station to load the raw part to be manufactured. The loading one is a delay station, hence, an instance of the random variable describing its fixed delay time is generated, and a departure event is scheduled to happen after that period. The system's state and statistics are updated as follows

```
total_cycle_time += (clock - node_event->event.arrival_load);
total_response_time += (clock - node_event->event.start_man);
node_event->event.rem_service_time = 0.0;
node_event->event.start_man = 0.0;
node_event->event.num_visits_M1 = 0;
LOAD.nsys++;
LOAD.narr++;
```

where `node_event->event.arrival_load` stores the time of the arrival of the AGV at the loading station, while `node_event->event.start_man` stores the time of the first arrival at the M1 station, which marks the start of the manufacturing process. `node_event->event.num_visits_M1` counts the number of visits made by the AGV to the M1 station during the manufacturing process of single part (it will be greater than one in case the grinding of the part is interrupted by maintenance).

departure_load Departure of the AGV after being loaded with the raw part to be manufactured. Classical departure from a delay station, with an arrival to M1 scheduled to take place immediately. The following code updates state and statistics:

```
 nPartsStarted++;
system_sum.service += node_event->event.service_time;
LOAD.station_sum.delay += node_event->event.service_time;
LOAD.station_sum.wait += node_event->event.service_time;
LOAD.ncomp++;
LOAD.nsys--;
```

arrival_M1 Arrival of an AGV to the M1 station to start (or resume after the previous interruption) the grinding of the carried part. The whole handler is discussed in the following section.

departure_M1 Departure of an AGV from the M1 station after the (whole or partial) completion of part elaboration. The whole handler is discussed in the following section.

arrival_M2 Arrival of an AGV to the M2 station, for the next step of the manufacturing of the carried part. The event is handled as a standard arrival to an SSQ station, with a departure being immediately scheduled if only one customer is in the station. Otherwise, the event is added to the input queue of the station. System's state is updated as follows

```
M2.nsys++;
M2.narr++;
```

departure_M2 Departure from the M2 station. Again, the handler is the standard for a departure from an SSQ station, with a departure of the following job waiting in queue (if any) being scheduled, as well as the arrival of the current job to either the M3 or the UNLOAD station, according to the routing probabilities of the project specification. System's state and statistics are updated as follows

```
M2.nsys--;
M2.ncomp++;

system_sum.service += node_event->event.service_time;
M2.station_sum.service += node_event->event.service_time;
wait = clock - node_event->event.arrival_time;
system_sum.wait += wait;
M2.station_sum.wait += wait;

if (M2.nsys > 0) {
    delay = clock - next_job->event.arrival_time;
    system_sum.delay += delay;
    M2.station_sum.delay += delay;
}
```

arrival_M3 Arrival of the AGV to the M3 machine. The same logic used for arrival_M2 is employed. The system's state and statistics are updated as follows

```
M3.nsys++;
M3.narr++;
node_event->event.create_time = clock;
node_event->event.service_time = GetServiceM3();
```

departure_M3 Departure of the AGV from machine M3 after it has completed its processing of the part. The same logic of `departure_M2` handler applies here, but no probabilistic choice is required, and arrival to M1 is immediately scheduled. The system's state and statistics are updated by this code

```
M3.nsys--;
M3.ncomp++;

system_sum.service += node_event->event.service_time;
M3.station_sum.service += node_event->event.service_time;
wait = clock - node_event->event.arrival_time;
system_sum.wait += wait;
M3.station_sum.wait += wait;

if (M3.nsys > 0) {
    delay = clock - next_job->event.arrival_time;
    system_sum.delay += delay;
    M3.station_sum.delay += delay;
}
```

arrival_unload The arrival of the AGV to unload the manufactured part. The same approach used to handle the arrival to the loading station is used. The system's state and statistics are updated as follows

```
UNLOAD.narr++;
UNLOAD.nsys++;

nPartsCompleted++;
total_manufacturing_time += (clock - node_event->event.start_man);
if (node_event->event.isTagged){
    manufacturing_time_sum += (clock - node_event->event.start_man);
    passages_counter++;
}
```

where `node_event->event.isTagged` memorizes whether the event under consideration refers to the AGV selected as tagged customer. In that case, we update the variables that will form the result of the regeneration cycle.

departure_unload Departure of the AGV after having unloaded the finished part. A uniform random variable instance is generated, and its value is interpreted according to the routing probabilities listed in the project specification, to decide whether to schedule an arrival to the loading or to the recharge station, which will take place immediately. The system's state and statistics are updated as follows

```
UNLOAD.station_sum.delay += node_event->event.service_time;
```

```

UNLOAD.station_sum.wait += node_event->event.service_time;
UNLOAD.nsys--;
UNLOAD.ncomp++;

```

arrival_recharge The arrival of the AGV to the station for recharging its electric battery. From the handling perspective, it's a classical arrival to an SSQ station, with the system's state and statistics updated as follows

```

REC.nsys++;
REC.narr++;

```

departure_recharge Departure of the AGV after having recharged its battery. Standard departure from an SSQ station, with the system's state and statistics updated as follows

```

REC.nsys--;
REC.ncomp++;

system_sum.service += node_event->event.service_time;
REC.station_sum.service += node_event->event.service_time;

wait = clock - node_event->event.arrival_time;
system_sum.wait += wait;
REC.station_sum.wait += wait;

if (REC.nsys > 0) {
    delay = clock - next_job->event.arrival_time;
    system_sum.delay += delay;
    REC.station_sum.delay += delay;
}

```

print_state Event introduced to print the system's state after the first 30 events. It is scheduled, to happen immediately, at the end of the engine code when `event_counter == 30`. Its service time is zero to avoid any impact on performance evaluation.

1.3 Relevant Events Handlers

The most significant event handlers are those for the arrival and departure from the M1 station. This is consistent with that station exhibiting the most complex behaviour (it's really the component of the system that makes simulation the sole adequate approach).

1.4 Simple vs Extended Version

In order to achieve maximum flexibility when switching between the simplified and the extended version of the simulator, the EXTENDED macro, defined in the NESsq.h header, has been introduced. Switching between the two versions of the simulator is then achieved in the following way:

```
if (EXTENDED)
    // code for the extended version
else
    // code for the simplified example
```

1.5 Regenerative Method

The regenerative method has been implemented following the skeleton suggested for Homework 7. The two most relevant methods of this implementation are now briefly discussed. The RegPoint method, which checks whether a regeneration point has been reached, and is called on every event extracted from the future event list, contains the following code

```
bool RegPoint(event_notice ev) {
    bool is_reg_point = false;

    if (EXTENDED)
        is_reg_point = ev.type == ARRIVAL_M1
            && M1.nsys == 0
            && ev.num_visits_M1 == 0
            && ev.isTagged;
    else
        is_reg_point = ev.type == ARRIVAL_M1
            && ev.num_visits_M1 == 0
            && ev.isTagged;

    if (is_reg_point) {
        reg_samp.num_cycles++;
        num_obs--;
        if (reg_samp.num_cycles > 1)
            CollectRegStatistics();
    }

    return is_reg_point;
}
```

As anticipated before, the tagged customer approach has been followed. Hence, the regeneration point is the arrival of the the customer of interest and the entrance of the manufacturing subsystem (M1 station), when no service operations with non-negative exponential distribution are underway. The tagged customer

is the AGV 0, hence, when scheduling the arrival to the loading station in the initialization loop, the following instruction is executed:

```
curr_notice->event.isTagged = (i == 0)
```

When considering the simplified version, the service time of the M1 station is negative exponential, hence, the service of a job can be safely interrupted. However, when considering the extended version, the service time follows a hyper-exponential distribution, hence, we have a regeneration point only when M1 station is idling. This is a stricter condition, which means that regeneration points will be rarer, which explains why, on average, the simulation of the extended model tends to last longer. The CollectRegStatistics method collects the values needed to compute the statistics of interest for the regeneration cycle that has just concluded. The DecideToStop method implements the Sequential Stopping Rule to decide whether the simulation should be stopped and how many iterations still have to be made to reach the target precision. Its implementation is the following

```
bool DecideToStop(){
bool isOver = false;

if (num_obs == 0){
    ComputeConfidenceInterval();

    if (reg_samp.precision <= PRECISION + 0.0005){
        Stop = clock;
        isOver = true;
    } else {
        num_obs = floor(pow((PERCENTILE * reg_samp.stdev)
/ (reg_samp.mi * PRECISION), 2)) - reg_samp.num_cycles;
    }
}

return isOver;
}
```

The num_obs variable, which stores the number of observations still needed to reach the target precision, is initialized to 41, to ensure that the percentiles of the standard normal distribution (1.96, in our case) can be used. Target and actual precision are real numbers, hence equality within 0.0005 of each other is tested.

2 Simulator Validation process

2.1 Process structure

In order to verify the correctness of the simulator implementation, the following steps, aimed at verifying the coherence between its results and the theoretical ones, were followed:

- Verification of Operational analysis laws on both the simplified and the extended version of the simulator
- Bottleneck analysis of the system and computation of a numerical reference solution using the MVA algorithm
- Checking coverage of the average manufacturing time reference value by the confidence interval

2.2 Verification of Operational Analysis Laws

For each of the following laws, the quantity measured during the simulation was compared with the one obtained by applying the law, both for the extended and for the simplified models



- Utilization law $U = XS$
- Waiting Time Law $\bar{n} = X\bar{w}$
- Little's Formula $\bar{n} = \lambda\bar{w}$

In both cases, all the results were sufficiently similar.

2.3 Bottleneck Analysis and MVA Solution

A bottleneck analysis was later performed, aiming to assess the asymptotic behaviour of the system, as well as to check the correctness of the MVA solution. The system under discussion is closed, hence, the loading station can be taken as a reference station for the computation of performance measures. By solving $V = VQ$ using Gauss elimination, the following visits counts were obtained



$$V = \begin{bmatrix} 1 \\ 12.5 \\ 0.25 \\ 1 \\ 0.4 \end{bmatrix}$$



As a side consistency check, one can observe that these visit counts respect the routing probabilities of the project specification. Computing the service demands shows that the bottleneck station is M2.

$$D = \begin{bmatrix} 112.5 \\ 150 \\ 20 \\ 40 \\ 50 \end{bmatrix}$$

M1 and M2 are the most heavily utilized stations, which is normal, given that all parts must be processed (possibly multiple times) from both of them. All the manufacturing time quantities can be computed by observing that only stations

1, 2 and 3 are involved in the manufacturing process. A lower bound for response time (blue line) can be computed as follows

$$R(1) = D[1] + D[2] + D[3] + D[4] + D[5] = 372.5s \quad (3)$$

By observing that manufacturing time is determined only by stations M1, M2 and M3, a lower bound can be obtained in the following way

$$M(1) = R(1) - D[4] - D[5] = 282.5s \quad (4)$$

The saturation point (horizontal red line) for the response time can be derived as

$$N^* = \frac{R(1)}{D[2]} + \frac{Z_l}{D[2]} \approx 2.883jobs \quad (5)$$

The saturation point for the manufacturing time can be derived by observing that

$$M(n) = n(D[2] - D[4] - D[5]) + Z_l \quad (6)$$

from which we derive

$$N^* = \frac{M(1) - Z_l}{D[2] - D[4] - D[5]} \approx 3.7083jobs \quad (7)$$

The upper bound for the response time (green line) can be computed as follows

$$R(n) = n * D[2] - Z_l \quad (8)$$

while that of the manufacturing time is defined as

$$M(n) = R(n) - R_4(n) - R_5(n) = n(D[2] - D[4] - D[5]) + Z_l \quad (9)$$

The ideal throughput value can be computed as

$$X_0(n) = \frac{n}{R_0(1) + Z_l} \quad (10)$$

The effective bound is

$$X_{max} = \frac{1}{150} \quad (11)$$

and the saturation point is

$$n = \frac{R_0(1) + Z_l}{D_2} \approx 2.8833jobs \quad (12)$$

The MVA algorithm was later applied, to compute a numerical solution for traffic levels (i.e. number of AGVs in the system) ranging between 1 and 30. The algorithm's implementation follows the specification of Homework Four and computes the average manufacturing time for each traffic level as the sum of mean times spent at stations 1, 2, and 3. In the following two plots, one for the average manufacturing time, and the other for the response time, both the results

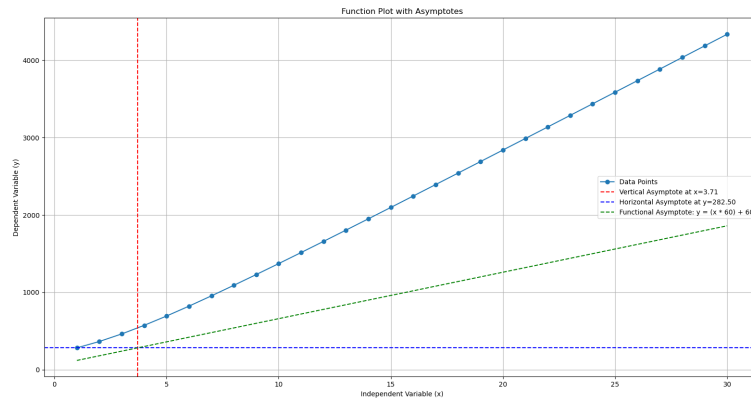


Fig. 1. Average Manufacturing Time

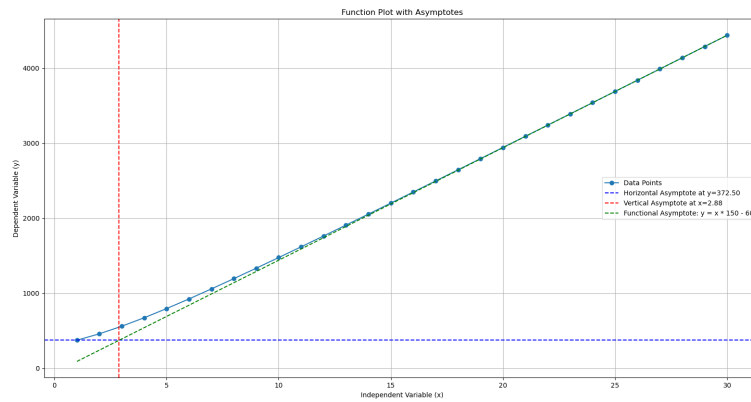


Fig. 2. Average Response Time

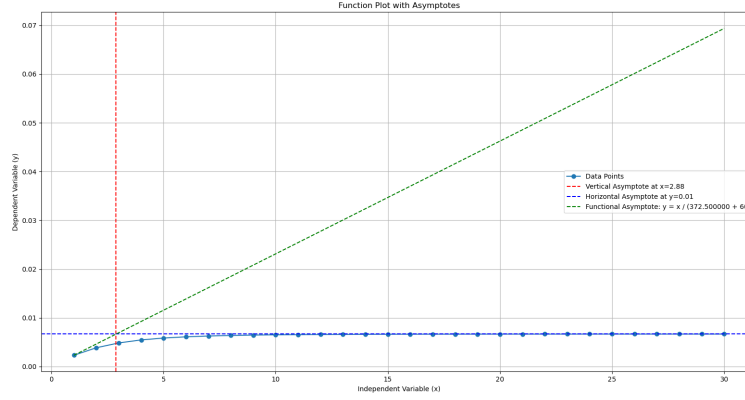


Fig. 3. Average Response Time

obtained with MVA (blue, solid line) and the asymptotes derived with bottleneck analysis (dashed lines) are shown, to highlight their coherence. Furthermore, coherence between performance values computed with MVA and measurements made using the simulator was manually checked for each traffic level, for both the simplified and the extended version. The sole detected inconsistency was on the waiting time (and, consequently, throughput) of the M1 station in the extended version of the system. The simulated value was higher than the numerical one, but this is normal, given that the simulator was using a different distribution for grinding time and different routing probabilities, as required by the project specification.

2.4 Statistical Validation

The project was statistically validated with the following two steps

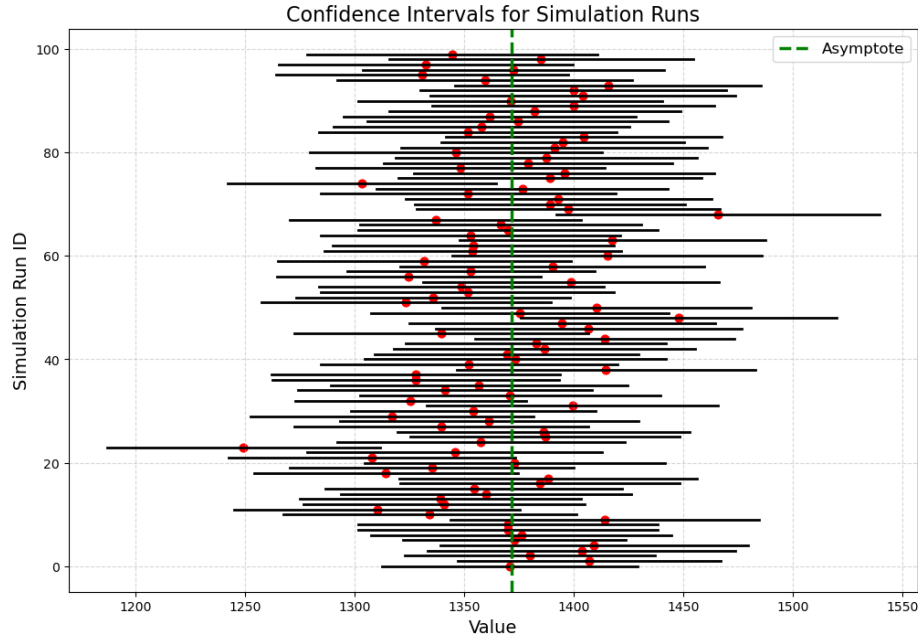
- Checking that confidence intervals cover MVA values
- Applying the definition of confidence level

As part of step one, average manufacturing time confidence intervals for a number of AGVs in the system between 1 and 30 were computed, using the simplified version of the simulator, with $\delta = 10^{10}$ and $seed = 123456789$. Results are reported in the following table, along with the corresponding MVA value.

We can notice that only one value is not covered by the corresponding confidence interval. This step only considers one specific realization of the confidence interval for each traffic level. Accounting for the variability in confidence intervals leads to the second validation step, which applies the definition of confidence level by repeating the simulation of a system with 10 AGVs 100 times, with different seeds. Each side was generated by applying the Python3 implementation

AGVs Number	Lower Bound	Upper Bound	MVA Value
1	267.323747	294.657633	282.500000000000
2	348.182915	372.715795	364.710982658960
3	442.835118	479.315299	462.551569381725
4	560.909184	620.565971	573.379714578224
5	657.391935	727.090497	694.054465230313
6	802.254601	877.124231	821.817962749463
7	897.398762	992.093781	954.625183127077
8	1005.058407	1111.283729	1091.083947460062
9	1219.305719	1346.604201	1230.269309529665
10	1312.272789	1429.028648	1371.555634279796
11	1410.005332	1531.349121	1514.501985261158
12	1651.854490	1825.195287	1658.782592729702
13	1721.568069	1900.468107	1804.146559883154
14	1945.098592	2136.921316	1950.394580589428
15	1890.805466	2081.642501	2097.365102759635
16	2142.069239	2366.994915	2244.925661760043
17	2164.515964	2390.091176	2392.967065228326
18	2452.601586	2682.948731	2541.399195371418
19	2636.206340	2910.762617	2690.147777114140
20	2840.779054	3116.563319	2839.151767830238
21	2856.902664	3157.583535	2988.361184506032
22	3079.312930	3404.898584	3137.735266313811
23	3260.290171	3604.795562	3287.240911854217
24	3260.290171	3604.795562	3436.851350613917
25	3467.533905	3826.494775	3586.545018021423
26	3442.159554	3752.400800	3736.304608388832
27	3667.860094	3992.942911	3886.116282804572
28	3872.771939	4263.514955	4035.969011041259
29	4035.027675	4463.404820	4185.854028350522
30	3996.624632	4384.113506	4335.764389837943

of SHA256 algorithm to the string "12345_run_ID" and converting the result to an integer in the interval $[0, 2^{31} - 1]$, to make sure it represents a viable seed for our random number generator. SHA256 was chosen because it's a trusted technique to ensure maximum independence between the seeds of consecutive runs. The results are summarized in the following plot, which displays on the x-axis the confidence interval bounds, and, on the y-axis, the ID of the simulation run. Each red dot corresponds to the mean of the confidence interval. The vertical, green, asymptote shows the average manufacturing time computed with MVA for a system with 10 AGV. We notice that 4 out of the 100 runs produced a confidence interval that do not cover the MVA value, which is consistent with the definition of 95% significance level.

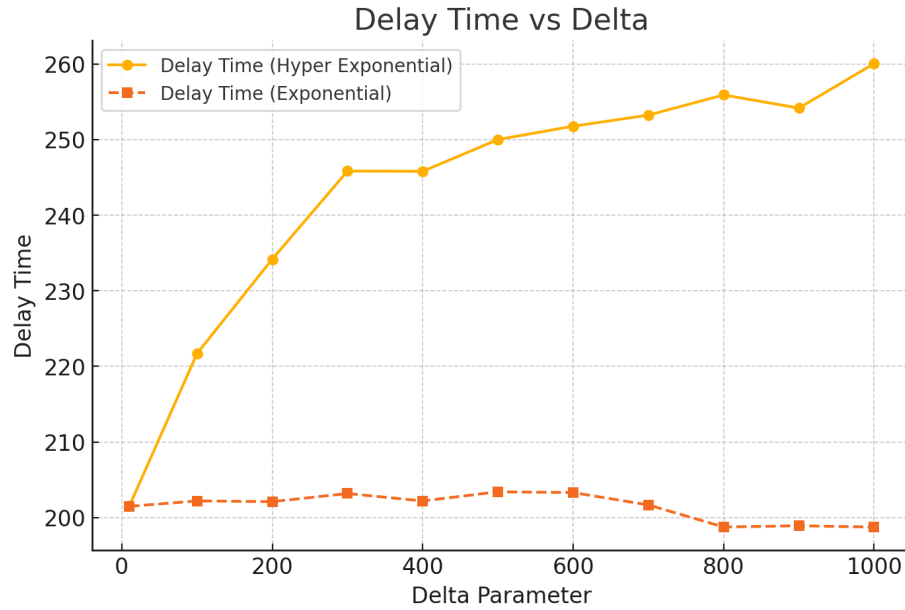


3 Systems Performance Analysis

After having validated it, the simulator was applied to analyze the performance of the manufacturing system. The analysis processes started by observing the results of the following runs (obtained with $seed = 123456789$) Analyzing these values, one can notice that switching to a higher-variance distribution (the hyper-exponential) has no relevant effect on the delay time of station M1, and, consequently, on average manufacturing time, when $\delta = 10$. On the other hand, the higher variance leads to higher average delay and manufacturing time when

δ	α	β	μ_1	μ_2	Average Delay M1	Average Manufacturing Time
10	0.8	0.2	50	250	201.511903	1373.7379
10	1	0	90	0	201.474872	1369.1174
10000	0.8	0.2	50	250	250.821629	1409.9537
10000	1	0	90	0	199.075119	1360.9358

$\delta = 10000$. This can be explained by observing that higher grinding time variance leads to bigger service times for the M1 station. Now, when δ is set to a small value, the manufacturing process of these jobs will be interrupted many times, giving shorter jobs the possibility of being scheduled without waiting too long in the station queue. Instead, when $\delta = 10000$, the service of these jobs will not be interrupted, leading shorter jobs to wait longer before being scheduled. In order to test this hypothesis, 10 simulation runs were conducted for both grinding time distributions, with δ values varying between 10 and 1000, and $seed = 123456789$. The results are shown in the following plot:



As expected, increasing δ also increases the delay (and, consequently, the manufacturing) time.