Computer Manufacturing Simulation. AI

Hugo Cortazar Alejo, Gonzalo Casado Cabezas, Hernán Fernández Gasalla

Contents

1 Introduction

In this project, we develop a simulation system for a computer production line, where each main component is produced independently. These components come from various specialized sub-factories, such as processors, graphics cards, storage units, RAM, power supplies, motherboards, cases, and cooling systems.

To carry out this simulation, we use Python and primarily the SimPy library, which allows modeling concurrent processes and limited resources. Each component is simulated through independent processes with variable manufacturing and transportation times. Additionally, the simulation includes the possibility of component failures with a predefined probability, triggering automatic retry mechanisms until success or until a maximum number of attempts is reached.

The project uses version control tools like GitHub, enabling effective code management and continuous progress tracking via a shared repository. The document includes development screenshots and detailed explanations of the project's modular structure, along with the results obtained in various simulated scenarios. This approach allowed us to identify potential improvements and assess the effectiveness of the implemented system.

2 System Structure

2.1 Directory Tree

```
Trabajo
├── main_fabric.py (Main factory, executes the general simulation)
├── transport.py (Component transportation)
├── components (Individual processes for each component)
|   ├── __init__.py (Empty file for initialization)
|   ├── box.py (Case manufacturing)
|   ├── cooling_system.py (Cooling system manufacturing)
|   ├── graphics_card.py (Graphics card manufacturing)
|   ├── mother_board.py (Motherboard manufacturing)
|   ├── power_supply.py (Power supply manufacturing)
|   ├── processor.py (Processor manufacturing)
|   ├── ram.py (RAM manufacturing)
|   └── storage.py (Storage manufacturing)
├── core (Additional system processes)
|   ├── __init__.py (Empty file for initialization)
|   └── final_assembly.py (Final assembly at the main factory) files)
├── logic (Logic and decision making processes)
|   ├── __init__.py (Empty file for initialization)
|   └── fuzzy_logic.py (Fuzzy logic decision-making system)
├── simulation_results.csv (Simulation output results in CSV format)
└── doc
    └── Computer Manufacturing Simulation (Documentation files)
```

Figure 1: Directory Tree

2.2 Functionality Overview

Main Factory

- Coordinates the simulation flow and component tracking across all processes.
- Launches and monitors all sub-factory processes and final assembly.
- Manages system load and resource constraints using SimPy.

Component Sub-factories

- Each module simulates the manufacturing of a specific hardware component (e.g., RAM, Storage).
- Handles variable production times and fuzzy-based assembly estimates.
- Incorporates probabilistic failures with retry limits for fault tolerance.

Final Assembly

- Oversees the final assembly once all required components are available.
- Verifies component completeness and handles success or failure of assembly.
- Logs completed product data into simulation results.

System Conditions

- Uses fuzzy logic to calculate dynamic assembly durations based on system conditions.
- Models human-like reasoning through fuzzy rules and membership functions.
- Enhances realism by making the assembly process context-sensitive.

Transportation

- Simulates delivery from component factories to the main factory with random delays.
- Adds realism through non-deterministic transport times.
- Ensures orderly storage and avoids duplicate component entries.

3 Code Design

The project code is structured using a modular and process-oriented approach. It is written in Python and uses the SimPy library to manage discrete event simulations.

The system is split into independent modules, each specializing in a particular component or function. This modularity simplifies maintenance, improves scalability, and supports team collaboration through Git version control.

3.1 Primary System Modules

main_fabric.py:

- Entry point of the program.
- Initializes the SimPy environment and shared resources.
- Defines transportation times.
- Orchestrates manufacturing and assembly processes.

components module:

- Simulates the production of each computer component.
- Manages variable production times and failure retry mechanisms.
- Key files: `processor.py`, `graphics_card.py`, `storage.py`, `ram.py`, etc.

transportation.py

- Manages transportation of components to main fabric.
- Adds random delays.
- Calculates the total time taken.

core module:

- Manages final assembly processes.
- File: `final_assembly.py`.

logic module:

- Uses Scikit-Fuzzy to implement the logic systems.
- Defines fuzzy sets, fuzzy operations and build fuzzy interface systems for decision-making and control applications
- File: `fuzzy_logic.py`.

3.2 Used Libraries

- **simPy**: Powers the discrete event simulation framework to model process flows and resources.
- **random**: Used to generate random delays and simulate failures in manufacturing and transport.
- **pandas**: Used to store, process, and export simulation results in a structured tabular format.

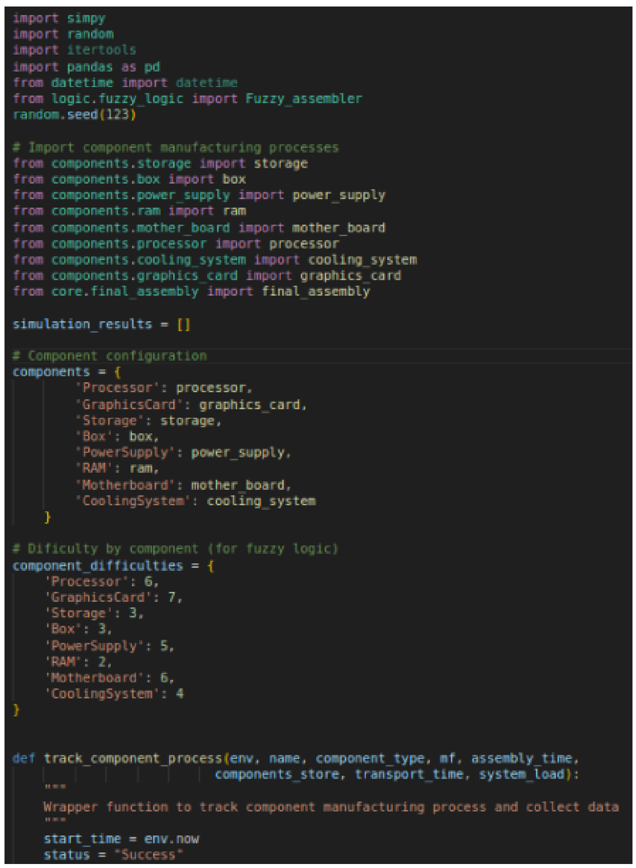
- **itertools**: Helps iterate over component combinations or control loops more efficiently.
- **numpy**: Provides numerical ranges and operations for defining fuzzy logic universes.
- **skfuzzy**: Implements fuzzy logic to calculate variable assembly times based on difficulty and load.

3.3 Version Control

- **GitHub**: GitHub Repository

4 Code Overview

main_fabric.py:



```python
import simpy
import random
import itertools
import pandas as pd
from datetime import datetime
from logic.fuzzy_logic import Fuzzy_assembler
random.seed(123)

# Import component manufacturing processes
from components.storage import storage
from components.box import box
from components.power_supply import power_supply
from components.ram import ram
from components.mother_board import mother_board
from components.processor import processor
from components.cooling_system import cooling_system
from components.graphics_card import graphics_card
from core.final_assembly import final_assembly

simulation_results = []

# Component configuration
components = {
        'Processor': processor,
        'GraphicsCard': graphics_card,
        'Storage': storage,
        'Box': box,
        'PowerSupply': power_supply,
        'RAM': ram,
        'Motherboard': mother_board,
        'CoolingSystem': cooling_system
    }

# Dificulty by component (for fuzzy logic)
component_difficulties = {
    'Processor': 6,
    'GraphicsCard': 7,
    'Storage': 3,
    'Box': 3,
    'PowerSupply': 5,
    'RAM': 2,
    'Motherboard': 6,
    'CoolingSystem': 4
}


def track_component_process(env, name, component_type, mf, assembly_time,
                            components_store, transport_time, system_load):
    """
    Wrapper function to track component manufacturing process and collect data
    """
    start_time = env.now
    status = "Success"
```

Figure 2: Main 1

```python
try:
    # Get the correct process function
    process_func = components[component_type]

    # Calcular tiempo de ensamblaje con lógica difusa
    difficulty = component_difficulties[component_type]
    fuzzy = Fuzzy_assembler()
    main_assembly_time = fuzzy.get_assembly_time(difficulty, system_load)

    yield env.process(process_func(
        env, name, mf, assembly_time, components_store, transport_time, system_load
    ))
except Exception as e:
    status = f"Failed: {str(e)}"

end_time = env.now
total_time = end_time - start_time

# Store results
simulation_results.append({
    "Component": component_type,
    "Name": name,
    "Start_Time": start_time,
    "End_Time": end_time,
    "Total_Time": total_time,
    "Assembly_Time": assembly_time,
    "Main_Assembly_Time": main_assembly_time,
    "Transport_Time": transport_time,
    "Status": status
})

__name__ == '__main__':
env = simpy.Environment()
main_factory = simpy.Resource(env, capacity=1)
components_store = simpy.Store(env)

from itertools import product

component = list(components.keys())
transport_times = [2, 3, 4 ,5, 6, 7, 8]

SYSTEM_LOAD = random.randint(0,9)

# Create manufacturing processes USING THE TRACKING FUNCTION
combinations = list(product(component, transport_times))
```

Figure 3: Main 2

```python
for component_type in components:
    for i in range(60):  # <-- genera 5 unidades de cada componente
        env.process(track_component_process(
            env,
            f"{component_type}_{i}",
            component_type,
            main_factory,
            1 + i,  # o valor fijo si prefieres
            components_store,
            2 + (i % 3),  # algo de variación en el transporte
            SYSTEM_LOAD
        ))
        i+1



# Start final assembly process
env.process(final_assembly(env, main_factory, components_store, system_load=SYSTEM_LOAD, simulation_results=simulation_results))

# Run the simulation
env.run()

# Verify data collection
print(f"\nTotal de registros recolectados: {len(simulation_results)}")

# Create DataFrame if we have data
if simulation_results:
    df = pd.DataFrame(simulation_results)

    # Debug: show column names
    print("\nColumnas disponibles:", df.columns.tolist())

    # Calculate transport delay if possible
    required_cols = ['Transport_Time', 'End_Time', 'Start_Time',
                     'Assembly_Time', 'Main_Assembly_Time']
    if all(col in df.columns for col in required_cols):
        df['Transport_Delay'] = df['Transport_Time'] - (
            df['End_Time'] - df['Start_Time'] -
            df['Assembly_Time'] - df['Main_Assembly_Time']
        )

    # Save to CSV
    csv_filename = "simulation_results.csv"
    df.to_csv(csv_filename, index=False)
    print(f"\nCSV generado correctamente: {csv_filename}")
    print("\nPrimeras filas:")
    print(df.head())
else:
    print("\nError: No se recolectaron datos durante la simulación")
    print("Posibles causas:")
    print("1. Los procesos no llegaron a ejecutarse completamente")
    print("2. La función track_component_process no se llamó correctamente")
    print("3. La simulación terminó antes de recolectar datos")
```

Figure 4: Main 3

- Initializes the SimPy environment.
- Manages global simulation coordination.

```python
import random
from transport import transport
from logic.fuzzy_logic import Fuzzy_assembler
random.seed(123)


def box(env, name, mf, assembly_time, components_store, transport_time, system_load):
    """
    Simulates the manufacturing and transportation of a box.
    :param env: Simulation environment.
    :param name: Name of the box.
    :param mf: Main factory resource.
    :param assembly_time: Manufacturing time.
    :param main_assembly_time: Assembly time at the main factory.
    :param components_store: Components store.
    :param transport_time: Transport time to the main factory.
    """
    difficulty = 3
    fuzzy = Fuzzy_assembler()
    main_assembly_time = fuzzy.get_assembly_time(difficulty, system_load)

    max_retries = 3  # Retry limit in case of failure
    retries = 0

    # Request access to main factory
    print(f'{name}: All box parts arrive at time {round(env.now)}')

    while retries < max_retries:  # Retry on failure
        # Simulate manufacturing time
        yield env.timeout(assembly_time)

        # Simulate 10% failure probability
        if random.random() < 0.1:
            print(f'{name}: Manufacturing failure. Restarting... (Attempt {retries + 1})')
            retries += 1
            continue  # Restart manufacturing process

        # Continue if successful
        break  # Exit loop if no failurefrom fuzzy_logic import FuzzyAssembler
    else:
        print(f'{name}: Critical failure. Could not manufacture box.')
        return  # Exit if retry limit exceeded

    with mf.request() as req:
        yield req

        # Simulate main factory assembly time
        print(f'{name}: Starts box manufacturing at time {round(env.now)}')
        yield env.timeout(main_assembly_time)
        print(f'{name}: Finishes assembly at time {round(env.now)}')

        # Simulate transport to main factory
        arrival = yield env.process(transport(env, transport_time))
        delay = arrival - env.now + transport_time  # Calculate delay
        print(f'{name}: Arrived at Main Factory at time {arrival} (Delay: {delay:.2f})')



        # Check for duplicates in store
        if 'Box' not in components_store.items:
            yield components_store.put('Box')
            print(f'{name}: Box sent to component store at time {round(env.now, 2)}')
        else:
            print(f'{name}: Duplicate box. Ignoring...')
```

Figure 5: box.py

```python
import random
random.seed(123)
from transport import transport
from logic.fuzzy_logic import Fuzzy_assembler

def cooling_system(env, name, mf, assembly_time, components_store, transport_time, system_load):
    """
    Simulates the manufacturing and transportation of a processor.
    :param env: Simulation environment.
    :param name: Name of the processor.
    :param mf: Main factory resource.
    :param assembly_time: Manufacturing time.
    :param main_assembly_time: Assembly time at the main factory.
    :param components_store: Components store.
    :param transport_time: Transport time to the main factory.
    """
    difficulty = 4
    fuzzy = Fuzzy_assembler()
    main_assembly_time = fuzzy.get_assembly_time(difficulty, system_load)

    max_retries = 3  # Retry limit in case of failure
    retries = 0

    # Component arrival message BEFORE any errors
    print(f'{name}: All cooling system components arrive at time {round(env.now)}')

    while retries < max_retries:  # Retry on failure
        # Simulate manufacturing time
        yield env.timeout(assembly_time)

        # Simulate a 10% chance of failure
        if random.random() < 0.1:
            print(f'{name}: Manufacturing failure. Restarting... (Attempt {retries + 1})')
            retries += 1
            continue  # Restart manufacturing process

        # If no failure, proceed
        break  # Exit loop if successful
    else:
        print(f'{name}: Critical failure. Could not manufacture the cooling system.')
        return  # Exit if retry limit exceeded

    # Request access to the main factory
    with mf.request() as req:
        yield req

        # Simulate assembly time at the main factory
        print(f'{name}: Starts cooling system manufacturing at time {round(env.now)}')
        yield env.timeout(main_assembly_time)
        print(f'{name}: Finishes assembly at time {round(env.now)}')

        # Simulate transport to the main factory
        arrival = yield env.process(transport(env, transport_time))
        delay = arrival - env.now + transport_time  # Calculate delay
        print(f'{name}: Arrived at Main Factory at time {arrival} (Delay: {delay:.2f})')

        # Check for duplicates
        if 'CoolingSystem' not in components_store.items:
            yield components_store.put('CoolingSystem')
            print(f'{name}: Cooling system sent to warehouse at time {round(env.now, 2)}')
        else:
            print(f'{name}: Duplicate processor. Ignoring...')
```

Figure 6: cooling_system.py

```python
import random
random.seed(123)
from transport import transport
from logic.fuzzy_logic import Fuzzy_assembler

def graphics_card(env, name, mf, assembly_time, components_store, transport_time, system_load):
    """
    Simulates the manufacturing and transportation of a graphics card.
    :param env: Simulation environment.
    :param name: Name of the graphics card.
    :param mf: Main factory resource.
    :param assembly_time: Manufacturing time.
    :param main_assembly_time: Assembly time at the main factory.
    :param components_store: Components store.
    :param transport_time: Transport time to the main factory.
    """

    difficulty = 7
    fuzzy = Fuzzy_assembler()
    main_assembly_time = fuzzy.get_assembly_time(difficulty, system_load)

    max_retries = 3  # Retry limit in case of failure
    retries = 0

    # Request access to the main factory
    print(f'{name}: All graphics card components arrive at time {round(env.now)}')
    while retries < max_retries:  # Retry on failure
        # Simulate manufacturing time
        yield env.timeout(assembly_time)

        # Simulate 10% failure probability
        if random.random() < 0.1:
            print(f'{name}: Manufacturing failure. Restarting... (Attempt {retries + 1})')
            retries += 1
            continue  # Restart manufacturing process

        # Continue if successful
        break  # Exit loop if no failure
    else:
        print(f'{name}: Critical failure. Could not manufacture graphics card.')
        return  # Exit if retry limit exceeded

    with mf.request() as req:
        yield req

        # Simulate main factory assembly time
        print(f'{name}: Starts graphics card manufacturing at time {round(env.now)}')
        yield env.timeout(main_assembly_time)
        print(f'{name}: Finishes assembly at time {round(env.now)}')

        # Simulate transport to main factory
        arrival = yield env.process(transport(env, transport_time))
        delay = arrival - env.now + transport_time  # Calculate delay
        print(f'{name}: Arrived at Main Factory at time {arrival} (Delay: {delay:.2f})')




        # Check for duplicates in store
        if 'GraphicsCard' not in components_store.items:
                yield components_store.put('GraphicsCard')
                print(f'{name}: Graphics card sent to warehouse at time {round(env.now, 2)}')
        else:
                print(f'{name}: Duplicate graphics card. Ignoring...')
```

Figure 7: graphics_card.py

```python
import random
random.seed(123)
from transport import transport
from logic.fuzzy_logic import Fuzzy_assembler

def mother_board(env, name, mf, assembly_time, components_store, transport_time, system_load):
    """
    Simulates the manufacturing and transportation of a motherboard.
    :param env: Simulation environment.
    :param name: Name of the motherboard.
    :param mf: Main factory resource.
    :param assembly_time: Manufacturing time.
    :param main_assembly_time: Assembly time at the main factory.
    :param components_store: Components store.
    :param transport_time: Transport time to the main factory.
    """
    difficulty = 6
    fuzzy = Fuzzy_assembler()
    main_assembly_time = fuzzy.get_assembly_time(difficulty, system_load)

    max_retries = 3  # Retry limit in case of failure
    retries = 0

    # Request access to main factory
    print(f'{name}: All motherboard components arrive at time {round(env.now)}')

    while retries < max_retries:  # Retry on failure
        # Simulate manufacturing time
        yield env.timeout(assembly_time)

        # Simulate 10% failure probability
        if random.random() < 0.1:
            print(f'{name}: Manufacturing failure. Restarting... (Attempt {retries + 1})')
            retries += 1
            continue  # Restart manufacturing process

        # Continue if successful
        break  # Exit loop if no failure
    else:
        print(f'{name}: Critical failure. Could not manufacture motherboard.')
        return  # Exit if retry limit exceeded

    with mf.request() as req:
        yield req

        # Simulate main factory assembly time
        print(f'{name}: Starts motherboard manufacturing at time {round(env.now)}')
        yield env.timeout(main_assembly_time)
        print(f'{name}: Finishes assembly at time {round(env.now)}')

        # Simulate transport to main factory
        arrival = yield env.process(transport(env, transport_time))
        delay = arrival - env.now + transport_time  # Calculate delay
        print(f'{name}: Arrived at Main Factory at time {arrival} (Delay: {delay:.2f})')



        # Check for duplicates in store
        if 'Motherboard' not in components_store.items:
            yield components_store.put('Motherboard')
            print(f'{name}: Motherboard sent to component store at time {round(env.now, 2)}')
        else:
            print(f'{name}: Duplicate motherboard. Ignoring...')
```

Figure 8: mother_board.py

```python
import random
random.seed(123)
from transport import transport
from logic.fuzzy_logic import Fuzzy_assembler

def power_supply(env, name, mf, assembly_time, components_store, transport_time, system_load):
    """
    Simulates the manufacturing and transportation of a power supply.
    :param env: Simulation environment.
    :param name: Name of the power supply.
    :param mf: Main factory resource.
    :param assembly_time: Manufacturing time.
    :param main_assembly_time: Assembly time at the main factory.
    :param components_store: Components store.
    :param transport_time: Transport time to the main factory.
    """
    difficulty = 5
    fuzzy = Fuzzy_assembler()
    main_assembly_time = fuzzy.get_assembly_time(difficulty, system_load)

    max_retries = 3  # Retry limit in case of failure
    retries = 0

    # Request access to main factory
    print(f'{name}: All power supply components arrive at time {round(env.now)}')

    while retries < max_retries:  # Retry on failure
        # Simulate manufacturing time
        yield env.timeout(assembly_time)

        # Simulate 10% failure probability
        if random.random() < 0.1:
            print(f'{name}: Manufacturing failure. Restarting... (Attempt {retries + 1})')
            retries += 1
            continue  # Restart manufacturing process

        # Continue if successful
        break  # Exit loop if no failure
    else:
        print(f'{name}: Critical failure. Could not manufacture power supply.')
        return  # Exit if retry limit exceeded

    with mf.request() as req:
        yield req

        # Simulate main factory assembly time
        print(f'{name}: Starts power supply manufacturing at time {round(env.now)}')
        yield env.timeout(main_assembly_time)
        print(f'{name}: Finishes assembly at time {round(env.now)}')

        # Simulate transport to main factory
        arrival = yield env.process(transport(env, transport_time))
        delay = arrival - env.now + transport_time  # Calculate delay
        print(f'{name}: Arrived at Main Factory at time {arrival} (Delay: {delay:.2f})')



        # Check for duplicates in store
        if 'PowerSupply' not in components_store.items:
            yield components_store.put('PowerSupply')
            print(f'{name}: PowerSupply sent to component store at time {round(env.now, 2)}')
        else:
            print(f'{name}: Duplicate PowerSupply. Ignoring...')
```

Figure 9: power_supply.py

```python
import random
random.seed(123)
from transport import transport
from logic.fuzzy_logic import Fuzzy_assembler

def processor(env, name, mf, assembly_time, components_store, transport_time, system_load):
    """
    Simulates the manufacturing and transportation of a processor.
    :param env: Simulation environment.
    :param name: Name of the processor.
    :param mf: Main factory resource.
    :param assembly_time: Manufacturing time.
    :param main_assembly_time: Assembly time at the main factory.
    :param components_store: Components store.
    :param transport_time: Transport time to the main factory.
    """
    difficulty = 6
    fuzzy = Fuzzy_assembler()
    main_assembly_time = fuzzy.get_assembly_time(difficulty, system_load)

    max_retries = 3  # Retry limit in case of failure
    retries = 0

    # Component arrival message BEFORE any errors
    print(f'{name}: All processor components arrive at time {round(env.now)}')

    while retries < max_retries:  # Retry on failure
        # Simulate manufacturing time
        yield env.timeout(assembly_time)

        # Simulate a 10% chance of failure
        if random.random() < 0.1:
            print(f'{name}: Manufacturing failure. Restarting... (Attempt {retries + 1})')
            retries += 1
            continue  # Restart manufacturing process

        # If no failure, proceed
        break  # Exit loop if successful
    else:
        print(f'{name}: Critical failure. Could not manufacture the processor.')
        return  # Exit if retry limit exceeded

    # Request access to the main factory
    with mf.request() as req:
        yield req

        # Simulate assembly time at the main factory
        print(f'{name}: Starts processor manufacturing at time {round(env.now)}')
        yield env.timeout(main_assembly_time)
        print(f'{name}: Finishes assembly at time {round(env.now)}')

        # Simulate transport to the main factory
        arrival = yield env.process(transport(env, transport_time))
        delay = arrival - env.now + transport_time  # Calculate delay
        print(f'{name}: Arrived at Main Factory at time {arrival} (Delay: {delay:.2f})')

        # Check for duplicates
        if 'Processor' not in components_store.items:
            yield components_store.put('Processor')
            print(f'{name}: Processor sent to warehouse at time {round(env.now, 2)}')
        else:
            print(f'{name}: Duplicate processor. Ignoring...')
```

Figure 10: processor.py

```python
import random
from transport import transport
from logic.fuzzy_logic import Fuzzy_assembler

def ram(env, name, mf, assembly_time, components_store, transport_time, system_load):
    """
    Simulates the manufacturing and transportation of a RAM memory.
    :param env: Simulation environment.
    :param name: Name of the RAM memory.
    :param mf: Main factory resource.
    :param assembly_time: Manufacturing time.
    :param main_assembly_time: Assembly time at main factory.
    :param components_store: Components storage.
    :param transport_time: Transportation time to main factory.
    """
    difficulty = 2
    fuzzy = Fuzzy_assembler()
    main_assembly_time = fuzzy.get_assembly_time(difficulty, system_load)

    max_retries = 3  # Max retries in case of failure
    retries = 0

    # Request access to main factory
    print(f'{name}: All RAM components arrived at time {round(env.now)}')

    while retries < max_retries:  # Retry on failure
        # Simulate manufacturing time
        yield env.timeout(assembly_time)

        # Simulate 10% failure probability
        if random.random() < 0.1:
            print(f'{name}: Manufacturing failure. Restarting... (Attempt {retries + 1})')
            retries += 1
            continue  # Restart manufacturing process

        # Continue if no failure
        break  # Exit loop if successful
    else:
        print(f'{name}: Critical failure. RAM could not be manufactured.')
        return  # Exit function if max retries reached

    with mf.request() as req:
        yield req

        # Simulate main factory assembly
        print(f'{name}: RAM assembly started at time {round(env.now)}')
        yield env.timeout(main_assembly_time)
        print(f'{name}: Assembly completed at time {round(env.now)}')

        # Simulate transportation
        arrival = yield env.process(transport(env, transport_time))
        delay = arrival - env.now + transport_time
        print(f'{name}: Arrived at Main Factory at {arrival}(Delay: {delay:.2f})')

        # Check for duplicates
        if 'RAM' not in components_store.items:
            yield components_store.put('RAM')
            print(f'{name}: RAM sent to component store at time {round(env.now, 2)}')
        else:
            print(f'{name}: Duplicate RAM. Ignoring...')
```

Figure 11: ram.py

```python
import random
random.seed(123)
from transport import transport
from logic.fuzzy_logic import Fuzzy_assembler

def storage(env, name, mf, assembly_time, components_store, transport_time, system_load):
    """
    Simulates the manufacturing and transportation of storage.
    :param env: Simulation environment.
    :param name: Name of the storage component.
    :param mf: Main factory resource.
    :param assembly_time: Manufacturing time.
    :param main_assembly_time: Assembly time at the main factory.
    :param components_store: Components store.
    :param transport_time: Transport time to the main factory.
    """
    difficulty = 3
    fuzzy = Fuzzy_assembler()
    main_assembly_time = fuzzy.get_assembly_time(difficulty, system_load)

    max_retries = 3  # Retry limit in case of failure
    retries = 0

    # Request access to main factory
    print(f'{name}: All storage components arrive at time {round(env.now)}')

    while retries < max_retries:  # Retry on failure
        # Simulate manufacturing time
        yield env.timeout(assembly_time)

        # Simulate 10% failure probability
        if random.random() < 0.1:
            print(f'{name}: Manufacturing failure. Restarting... (Attempt {retries + 1})')
            retries += 1
            continue  # Restart manufacturing process

        # Continue if successful
        break  # Exit loop if no failure
    else:
        print(f'{name}: Critical failure. Could not manufacture storage.')
        return  # Exit if retry limit exceeded

    with mf.request() as req:
        yield req

        # Simulate main factory assembly time
        print(f'{name}: Starts storage manufacturing at time {round(env.now)}')
        yield env.timeout(main_assembly_time)
        print(f'{name}: Finishes assembly at time {round(env.now)}')

        # Simulate transport to main factory
        arrival = yield env.process(transport(env, transport_time))
        delay = arrival - env.now + transport_time  # Calculate delay
        print(f'{name}: Arrived at Main Factory at time {arrival} (Delay: {delay:.2f})')



        # Check for duplicates in store
        if 'Storage' not in components_store.items:
            yield components_store.put('Storage')
            print(f'{name}: Storage sent to component store at time {round(env.now, 2)}')
        else:
            print(f'{name}: Duplicate storage. Ignoring...')
```

Figure 12: storage.py

- Demonstrates failure logic and retry mechanism during production.

transport.py:

```python
import random
random.seed(123)

def transport(env, transport_time):
    """
    Simulates component transportation with random delays.
    :param env: Simulation environment
    :param transport_time: Base transportation time
    :return: Actual arrival time (rounded to 2 decimal places)
    """
    # Generate random delay between 0-2 time units
    delay = random.uniform(0, 2)
    total_time = transport_time + delay

    # Simulate transportation process
    yield env.timeout(total_time)

    # Return precise arrival time
    return round(env.now, 2)
```

Figure 13: Transport

- Simulates transport delays.

final_assembly.py:

```python
import simpy
import random
from logic.fuzzy_logic import Fuzzy_assembler
random.seed(123)

def final_assembly(env, main_factory, components_store, system_load, simulation_results):
    required_components = [
        'Processor', 'GraphicsCard', 'Storage',
        'Box', 'PowerSupply', 'RAM', 'Motherboard', 'CoolingSystem'
    ]
    computer_counter = 0  # Computer assembly counter

    while True:
        collected_components = []
        # Component collection phase
        while len(collected_components) < len(required_components):
            component = yield components_store.get()

            if component in collected_components:
                print(f'Duplicate {component} received at time {round(env.now)}. Ignoring...')
                continue

            collected_components.append(component)
            print(f'{component} for computer {computer_counter} arrived at assembly line at time {round(env.now)}')

        # Component verification
        if set(collected_components) == set(required_components):
            difficulty = 6
            fuzzy = Fuzzy_assembler()
            estimated_time = fuzzy.get_assembly_time(difficulty, system_load)
            print(f'Starting assembly of computer {computer_counter} at time {round(env.now)}. Estimated duration: (fuzzy): {estimated_time:.2f}')

            # Simulate random assembly failure (10% probability)
            if random.random() < 0.1:
                print(f'ERROR! Computer {computer_counter} assembly failed at time {round(env.now)}. Restarting...')
                collected_components.clear()
                continue  # Restart process without incrementing counter

            # Successful assembly simulation
            try:
                yield env.timeout(random.uniform(estimated_time - 1, estimated_time + 1))
                print(f'Computer {computer_counter} successfully assembled at time {round(env.now)}')
                simulation_results.append({
                    "Component": "Computer",
                    "Name": f"Computer_{computer_counter}",
                    "Start_Time": None,
                    "End_Time": env.now,
                    "Total_Time": None,
                    "Assembly_Time": estimated_time,
                    "Main_Assembly_Time": None,
                    "Transport_Time": None,
                    "Status": "Success"
                })
                computer_counter += 1
            except simpy.Interrupt:
                print(f'Assembly interrupted for computer {computer_counter}')

        else:
            missing = set(required_components) - set(collected_components)
            print(f'Missing components for computer {computer_counter} at time {round(env.now)}: {", ".join(missing)}')

        collected_components.clear()
```

Figure 14: Final Assembly

- Integrates all components into a finished product.

fuzzy_logic.py:

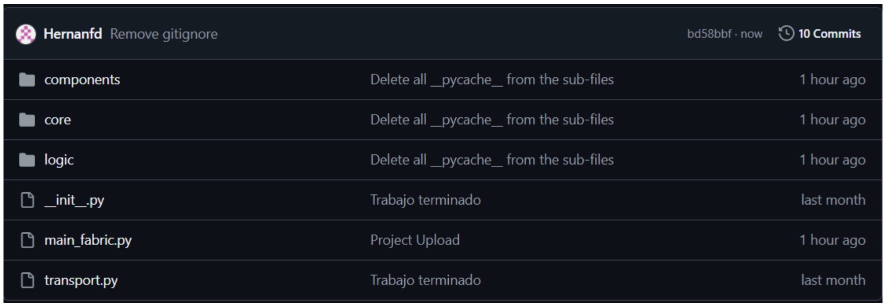
```python
import numpy as np
import skfuzzy as fuzz
from skfuzzy import control as ctrl

class Fuzzy_assembler:
    def __init__(self):
        # Inputs
        self.difficulty = ctrl.Antecedent(np.arange(0, 11, 1), 'difficulty')
        self.load = ctrl.Antecedent(np.arange(0, 11, 1), 'load')
        # Output
        self.assembly_time = ctrl.Consequent(np.arange(0, 11, 1), 'assembly_time')

        # Membership functions
        self.difficulty.automf(3, names = ["poor", "average", "good"])
        self.load.automf(3, names = ["poor", "average", "good"])

        self.assembly_time['short'] = fuzz.trimf(self.assembly_time.universe, [0, 0, 5])
        self.assembly_time['medium'] = fuzz.trimf(self.assembly_time.universe, [2, 5, 8])
        self.assembly_time['long'] = fuzz.trimf(self.assembly_time.universe, [5, 10, 10])

        # Rules
        rule1 = ctrl.Rule(self.difficulty['poor'] & self.load['poor'], self.assembly_time['short'])
        rule2 = ctrl.Rule(self.difficulty['good'] & self.load['average'], self.assembly_time['medium'])
        rule3 = ctrl.Rule(self.difficulty['good'] & self.load['good'], self.assembly_time['long'])
        rule4 = ctrl.Rule(self.difficulty['average'] | self.load['average'], self.assembly_time['medium'])

        # Control system
        self.controller = ctrl.ControlSystem([rule1, rule2, rule3, rule4])
        self.simulator = ctrl.ControlSystemSimulation(self.controller)

    def get_assembly_time(self, difficulty_value, load_value):
        self.simulator.input['difficulty'] = difficulty_value
        self.simulator.input['load'] = load_value
        self.simulator.compute()
        return self.simulator.output['assembly_time']
```

Figure 15: Logic

- The logic used in the system.

Repository (GitHub):

- Repository structure and collaborative version control management.



Hernanfd Remove gitignore		bd58bbf · now 10 Commits
components	Delete all __pycache__ from the sub-files	1 hour ago
core	Delete all __pycache__ from the sub-files	1 hour ago
logic	Delete all __pycache__ from the sub-files	1 hour ago
__init__.py	Trabajo terminado	last month
main_fabric.py	Project Upload	1 hour ago
transport.py	Trabajo terminado	last month

Figure 16: Github Repository

5 Simulation

```python
import pandas as pd

# Load the simulation results CSV
df = pd.read_csv("simulation_results.csv")  # Replace with your actual file
df.head()
```

	Component	Name	Start_Time	End_Time	Total_Time	Assembly_Time	Main_Assembly_Time	Transport_Time	Status	Transport_Delay
0	Processor	Processor_0	0.0	9.451656	9.451656	1.0	5.000000	2.0	Success	-1.451656
1	GraphicsCard	GraphicsCard_0	0.0	17.521497	17.521497	1.0	5.000000	2.0	Success	-9.521497
2	Storage	Storage_0	0.0	24.646424	24.646424	1.0	3.922533	2.0	Success	-17.723891
3	Box	Box_0	0.0	31.980454	31.980454	1.0	3.922533	2.0	Success	-25.057922
4	CoolingSystem	CoolingSystem_11	0.0	36.000000	36.000000	12.0	4.447732	4.0	Success	-15.552268

Figure 17: Simulation 1

```python
import matplotlib.pyplot as plt

# Calculate and plot average transport delay
df['Transport_Delay'] = df['Transport_Time'] - (
    df['End_Time'] - df['Start_Time'] - df['Assembly_Time'] - df['Main_Assembly_Time']
)

df.groupby('Component')['Transport_Delay'].mean().plot(kind='bar', title="Average Transport Delay by Component")
plt.ylabel("Transport Delay (time units)")
plt.xticks(rotation=45)
plt.grid(True)
plt.tight_layout()
plt.show()
```

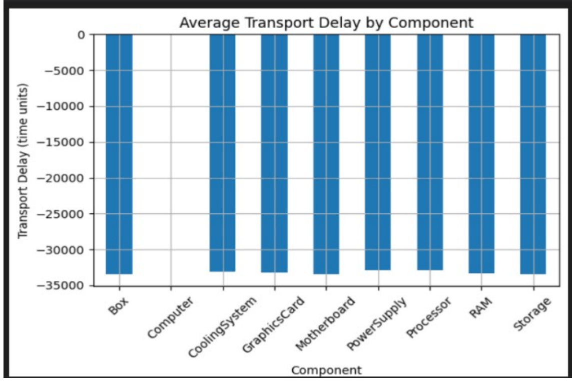


Figure 18: Simulation 2

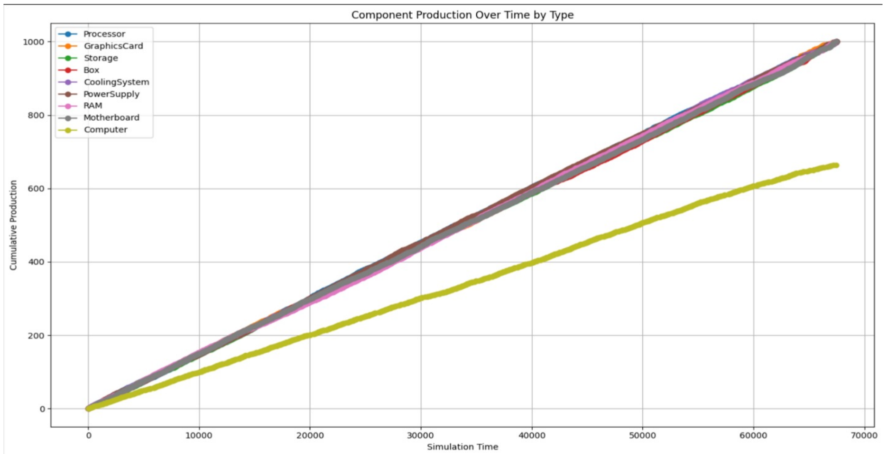


Figure 19: Simulation 3

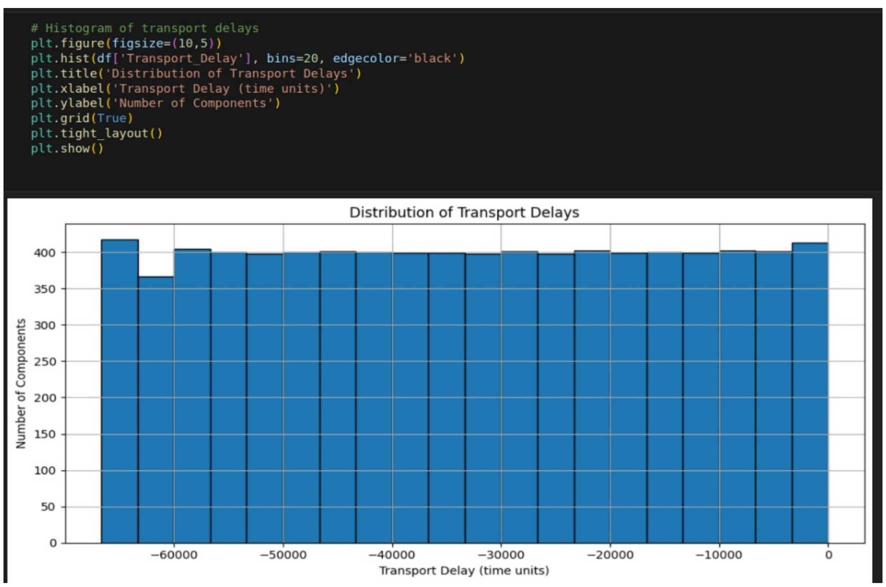


Figure 20: Simulation 4

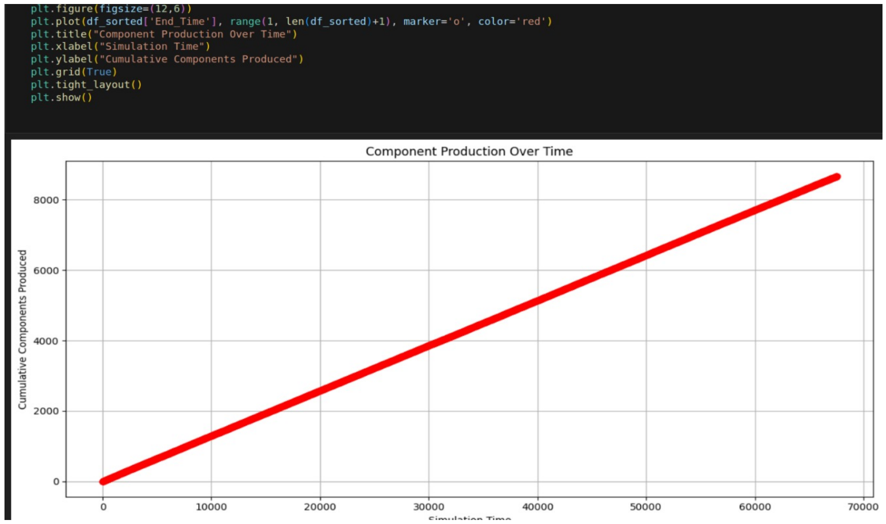


Figure 21: Simulation 5

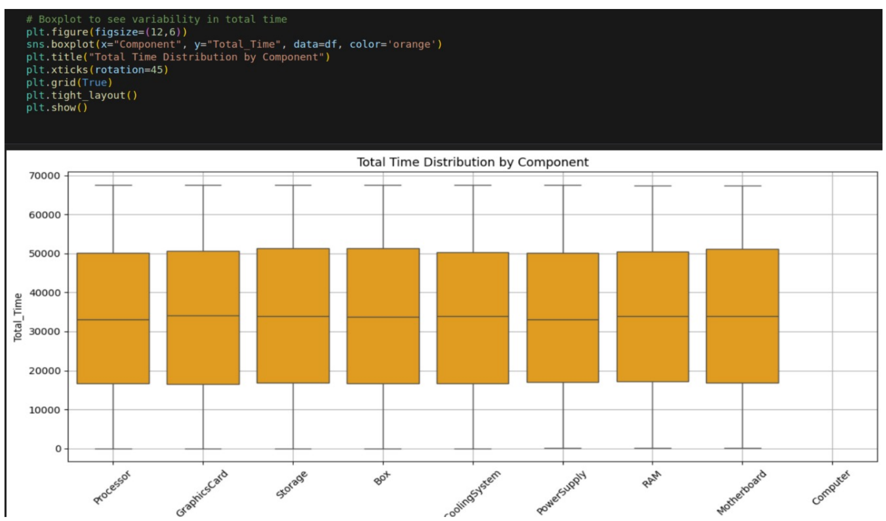


Figure 22: Simulation 6

```python
# Scatter plot of Assembly Time vs Transport Time
plt.figure(figsize=(10,6))
sns.scatterplot(data=df, x="Assembly_Time", y="Transport_Time", hue="Component")
plt.title("Assembly Time vs Transport Time")
plt.grid(True)
plt.tight_layout()
plt.show()
```

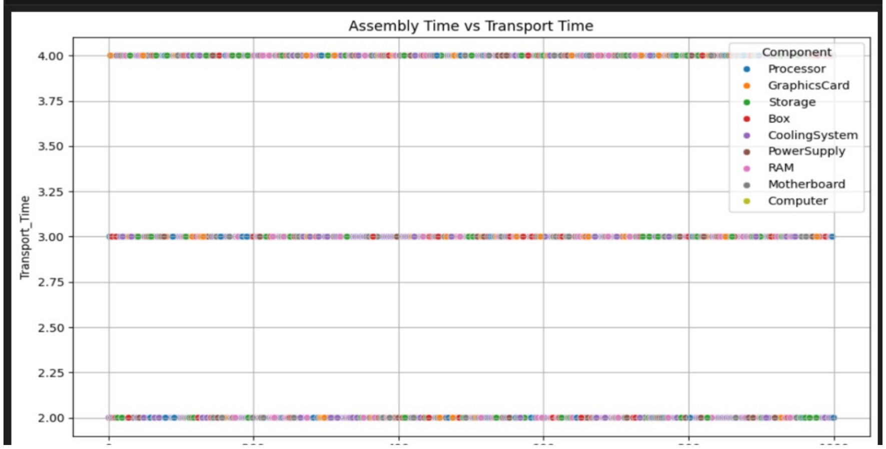


Figure 23: Simulation 7

```python
# Calculate correlation matrix
corr = df[['Total_Time', 'Assembly_Time', 'Transport_Time']].corr()

# Plot heatmap
plt.figure(figsize=(8,6))
sns.heatmap(corr, annot=True, cmap="coolwarm", linewidths=0.5)
plt.title("Correlation Matrix of Times")
plt.show()
```

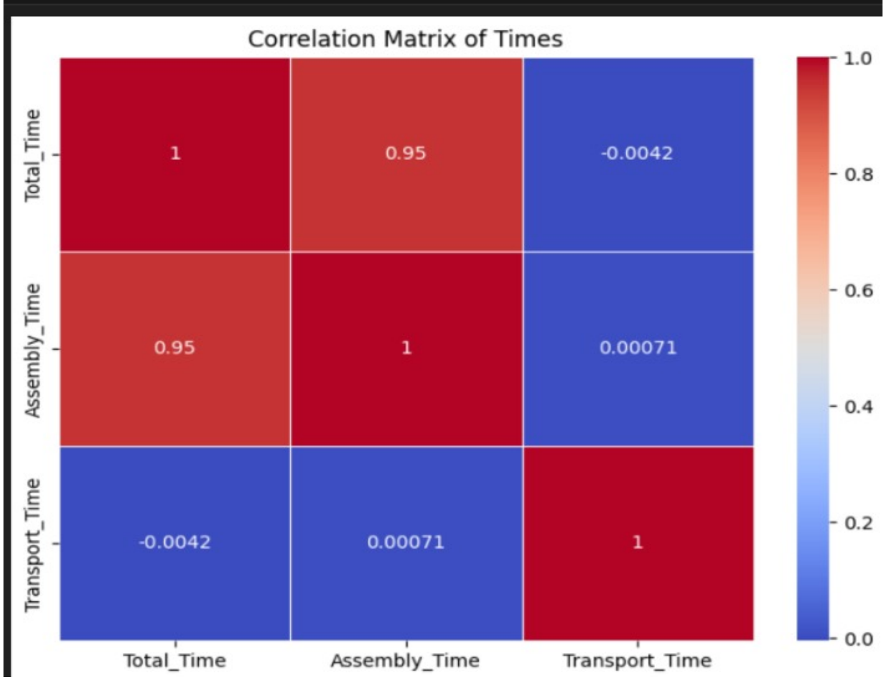


Figure 24: Simulation 8

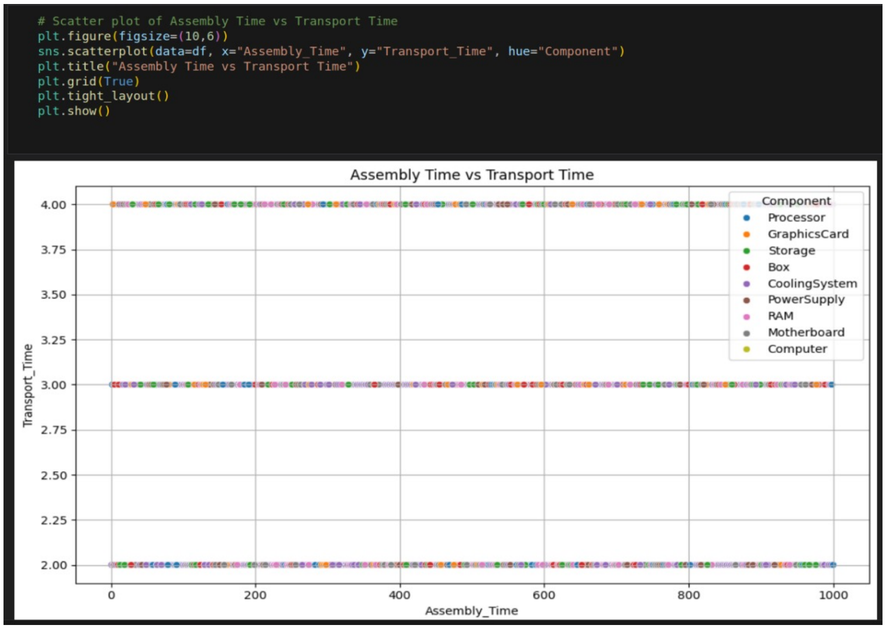


Figure 25: Simulation 9

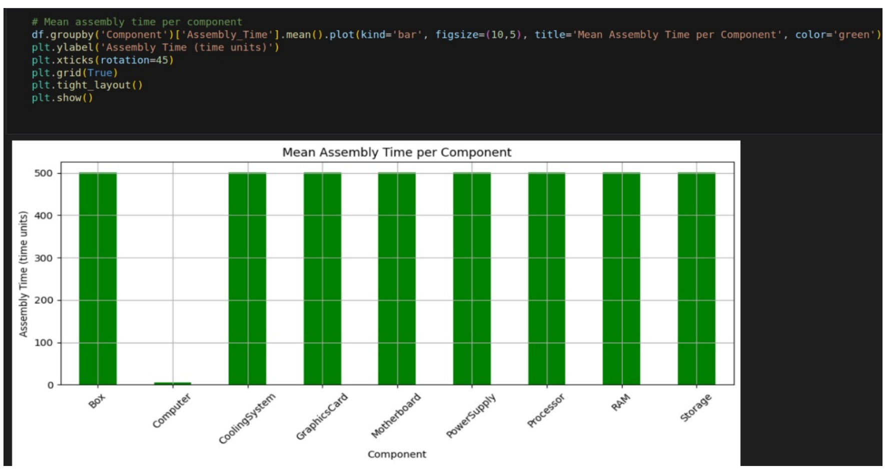


Figure 26: Simulation 10

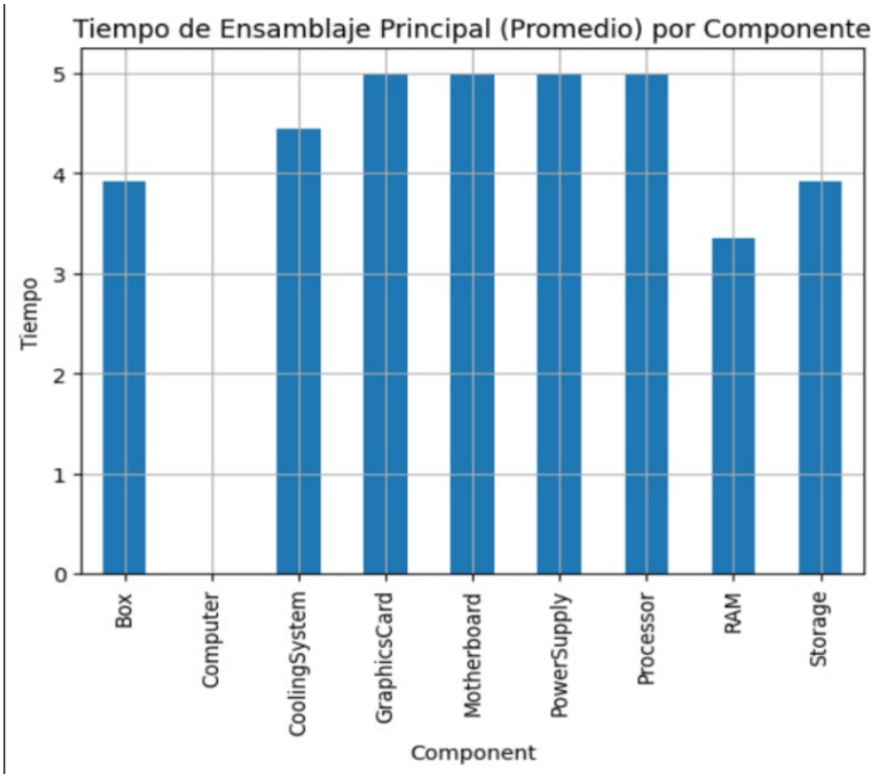


Figure 27: Simulation 11

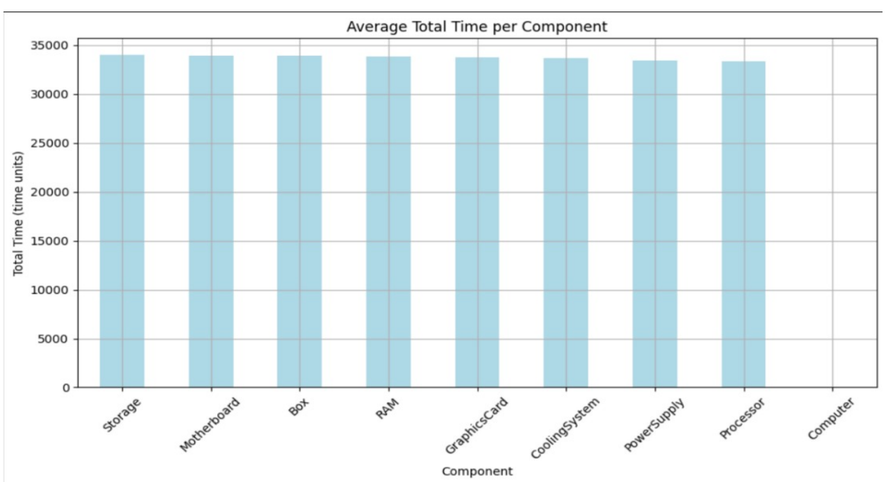


Figure 28: Simulation 12

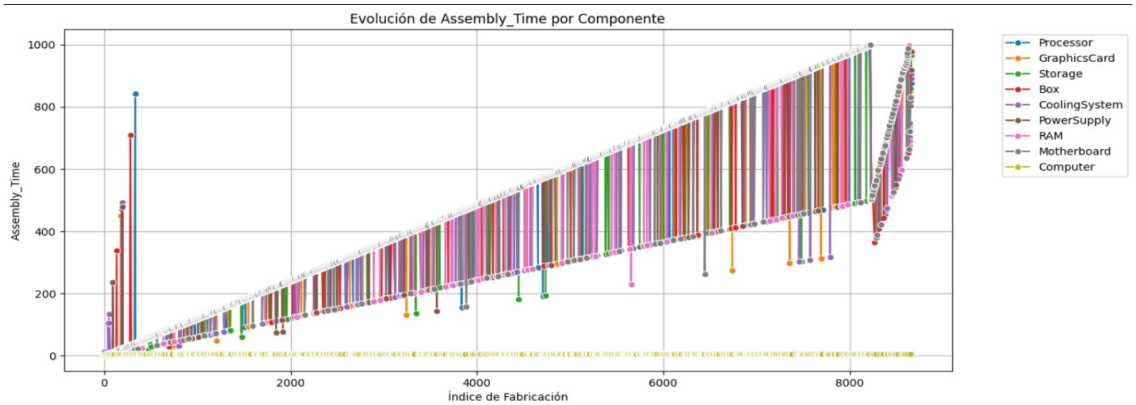


Figure 29: Simulation 13

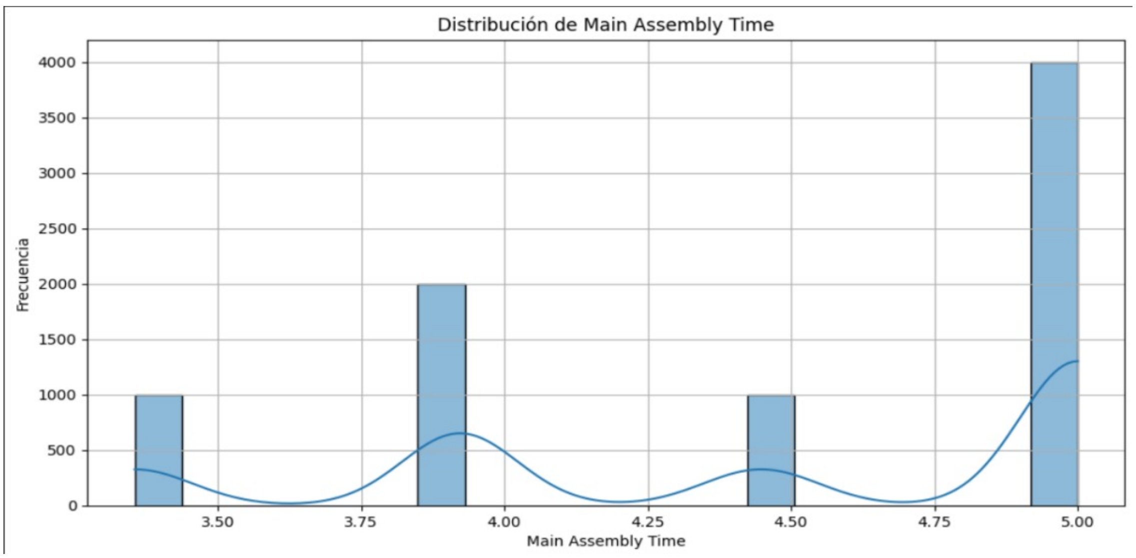


Figure 30: Simulation 14