Date: 14/04/2025

Course: CSE316

PROJECT REPORT

Project Overview

Project Title

“A Dynamic Load Balancer for Multiprocessor Systems”

Project Objective

The primary objective of this project is to design and implement an efficient dynamic load-balancing algorithm for multiprocessor systems to optimize resource utilization, minimize response time, and enhance overall system performance. The project aims to:

- Address the challenge of uneven workload distribution in multiprocessor environments.
- Improve system efficiency by dynamically redistributing tasks among processors.
- Reduce latency and prevent processor idle time, ensuring optimal performance under varying workloads.

Scope of Work

The project encompasses the development and implementation of a dynamic load balancer designed to optimize task distribution across multiprocessor systems. The key deliverables include:

1. **Dynamic Load Balancer with Real-time Monitoring and Automatic Rebalancing**: This component is responsible for intelligently assigning tasks to processors based on their current workload, temperature, and power consumption. It continuously monitors these metrics and redistributes tasks as needed to prevent bottlenecks and ensure optimal performance. The load balancer uses algorithms to prioritize tasks and make distribution decisions, taking into account the specific requirements of each task and the current state of each processor.

2. **Graphical User Interface (GUI)**: A user-friendly interface that visualizes processor load and statistics in real-time. The GUI allows users to monitor the system's performance, view detailed processor metrics, and interact with the load balancer. It provides a clear and intuitive way to understand how tasks are being distributed and how the system is responding to varying workloads.

3. **Comprehensive Documentation**: Detailed guides for setting up, using, and deploying the load balancer. This includes instructions for installation, configuration, and troubleshooting, as well as explanations of the underlying algorithms and design principles.

INCLUSION

The Inclusions in this project are the core functionalities that enable the load balancer to effectively manage tasks across processors. These include:

Task Prioritization and Distribution Algorithms: The logic that determines which tasks should be assigned to which processors, based on factors such as task type, processor specialization, and current system state.

Processor Health Monitoring: Real-time tracking of processor metrics such as CPU usage, memory usage, temperature, and power consumption. This data is used to make informed decisions about task distribution and rebalancing.

Load Rebalancing Mechanisms: The processes that trigger task redistribution when a processor becomes overloaded or when system conditions change. This ensures that no single processor is overwhelmed and that all resources are used efficiently.

EXCLUSION

The Exclusions from this project are features and functionalities that are beyond the current scope. These include:

Support for Non-Python Environments: The load balancer is designed to work within Python-based systems and does not extend to other programming environments.

Advanced Machine Learning Optimizations: While the load balancer uses intelligent algorithms for task distribution, it does not incorporate machine learning techniques for predictive task assignment or other advanced optimizations.

Who It's For

Primary Users: IT pros and system admins who want to optimize their hardware.

Secondary Users: Researchers, teachers, and businesses looking to squeeze more performance out of their systems.

Module-Wise Breakdown

1. Processor Queue Management

This module is like a **personal assistant** for each processor. It manages the task queue, ensures tasks are executed in order, and prevents overload. It tracks metrics like load, temperature, and power consumption, updating them as tasks come and go. Think of it as the processor's dedicated helper, keeping everything organized and running smoothly.

2. Dynamic Load Balancer

The **smart scheduler** of the system. It decides which processor handles each task based on current workload and specialization. If a processor gets too busy, it redistributes tasks to keep things balanced. It's always monitoring performance and adjusting in real-time to ensure optimal efficiency.

3. Task Submission and Management

This is the **system's receptionist**. It receives new tasks, submits them to the load balancer, and tracks their status—whether pending, processing, or completed. It ensures every task is accounted for and updates the system on what's happening.

4. Real-time Monitoring and Visualization

The **dashboard** that gives you a live view of your system. Through a GUI, you can see processor load, task completion rates, temperature, and power consumption. It's like a window into your computer's brain, letting you pause and inspect details whenever needed.

Each module works together to create a system that's efficient, transparent, and easy to use, giving you full control over how your computer manages its workload.

Functionalities

Task Distribution
Finds the best processor for each task based on what it's good at and how busy it is. Like assigning the right person to the right job.

Real-time Monitoring
Keeps an eye on processor load, temperature, and power use—like a constant health check to ensure everything's running smoothly.

Automatic Load Balancing
Redistributes tasks if a processor gets overwhelmed, ensuring no single processor is overworked. Think of it as adjusting workloads in a team to keep everyone productive.

Health Checks
Monitors processor health to prevent overheating or overuse, ensuring optimal performance—like giving your computer a regular checkup.

Visualization

Provides a live dashboard to see processor load and stats. It’s like having a window into your computer’s brain to see what’s happening in real-time.

load_balancer.py

This is the core implementation of the dynamic load balancer. It:

Monitors Processor Metrics: Tracks CPU usage, memory usage, temperature, and power consumption in real-time.

Intelligent Task Distribution: Assigns tasks to processors based on their current load, specialization, and health.

Automatic Load Rebalancing: Redistributes tasks when a processor becomes overloaded, ensuring no single processor is overwhelmed.

Supports Task Types: Handles compute-intensive, memory-intensive, I/O-intensive, and balanced tasks efficiently.

Predictive and Responsive: Uses a monitoring thread to continuously assess system state and adjust task assignments proactively.

simple_load_balancer_gui.py

This file provides a graphical interface to visualize the load balancer's operations:

Real-time Visualization: Displays processor load over time using a dynamic graph.

Processor Statistics: Shows detailed metrics like tasks processed, success rates, current load, temperature, and power consumption.

User Interaction: Allows users to adjust the number of processors and pause/resume updates for closer inspection.

Artificial Load Simulation: Generates synthetic load data to demonstrate how the system responds to varying workloads.

README.md

The documentation file explains the project's purpose, functionality, and usage:

Project Overview: Describes the problem of inefficient resource allocation and how the load balancer solves it.

Key Features: Highlights smart task distribution, real-time monitoring, automatic balancing, and health checks.

Getting Started: Provides clear instructions for setting up and running the project.

Educational Value: Positions the project as a learning tool for understanding system resource management and Python programming.

Technology Used

Programming Languages

Python 3.7 or newer: The primary programming language used for implementing the dynamic load balancer and the graphical user interface.

Libraries and Tools

psutil: A cross-platform library for retrieving information on running processes and system utilization (CPU, memory, etc.).

matplotlib: A plotting library used for creating static, animated, and interactive visualizations in the GUI.

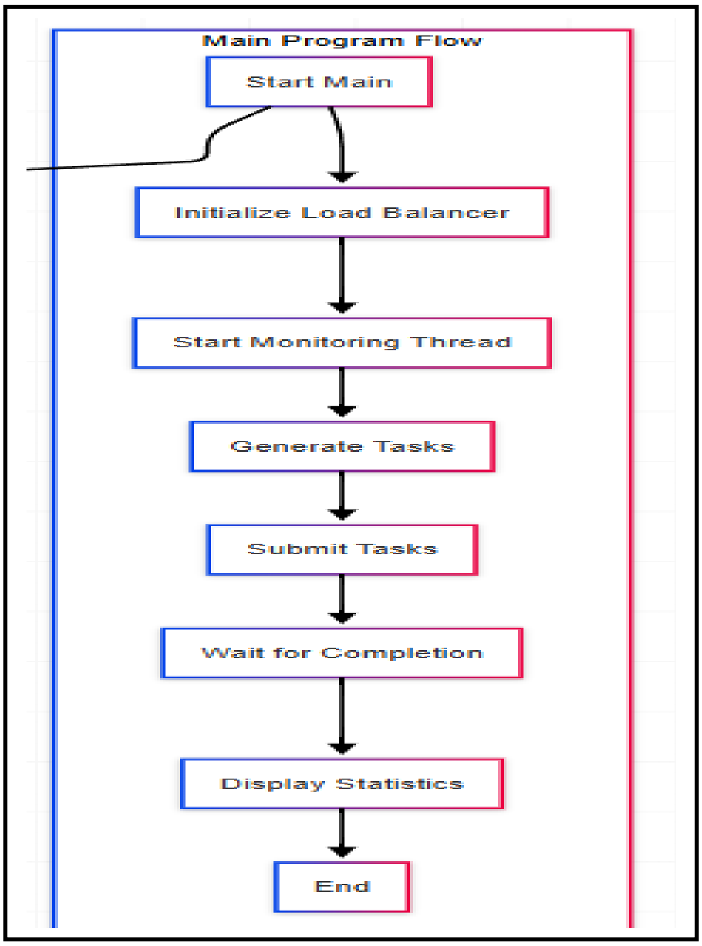
typing: A library that provides support for type hints, improving code clarity and helping Python understand the code better.

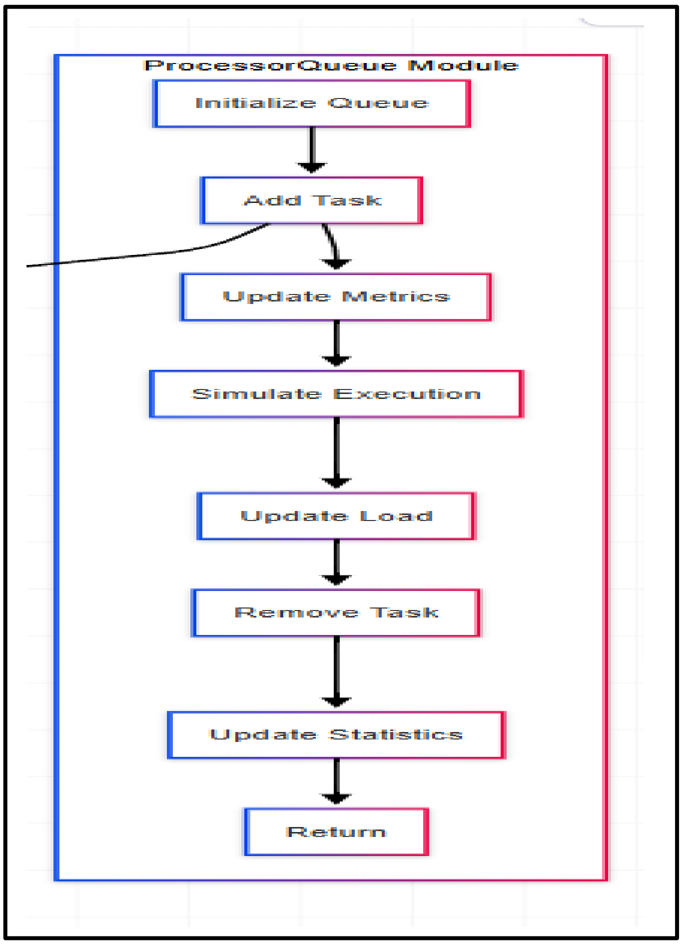
Other Tools

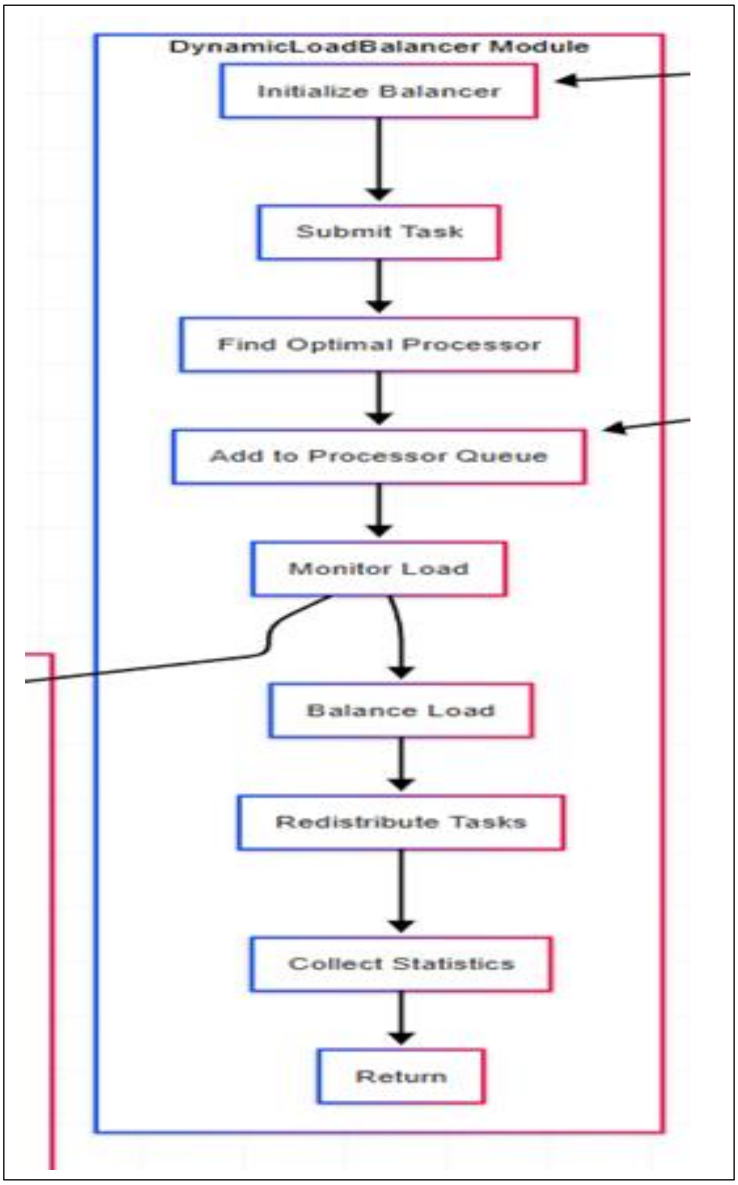
GitHub: Used for version control to manage the project's source code, track changes, and collaborate with team members.

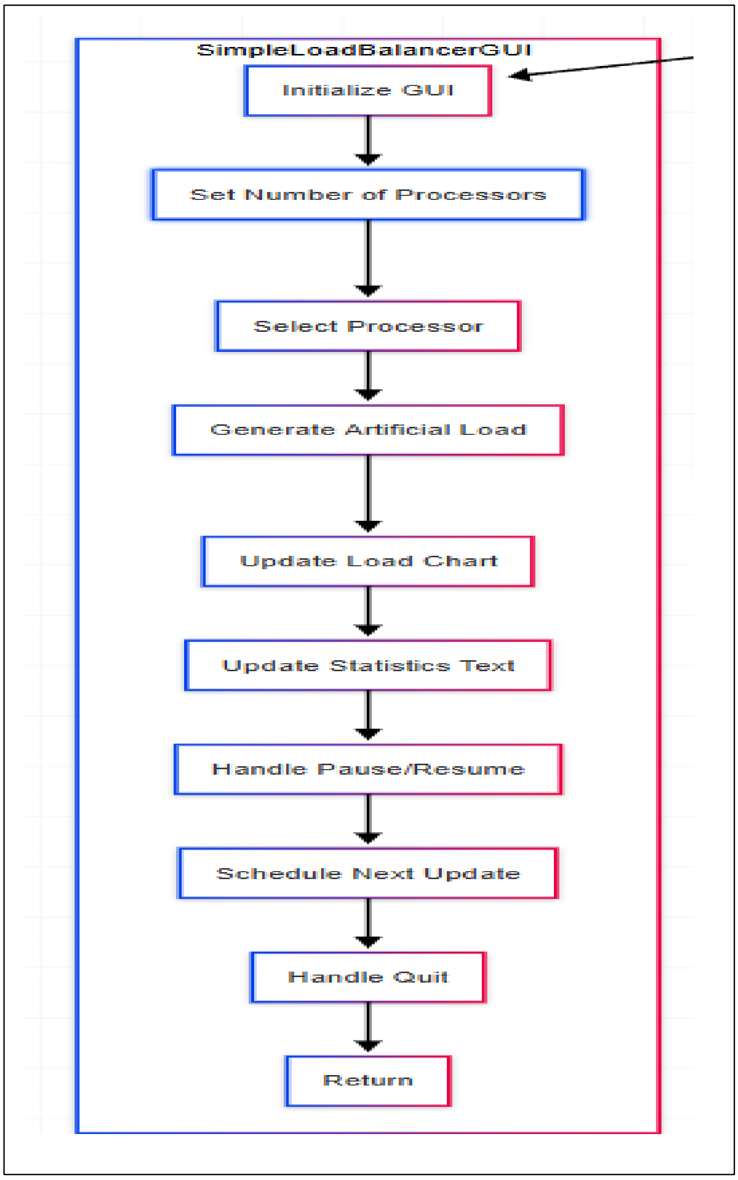
Flow Diagram

Mermaid Chart Link: https://www.mermaidchart.com/app/projects/1c0f0c91-c136-45f0-b77d-cb6e454f559c/diagrams/05499e7b-388c-43a5-8755-655b2d4c2603/version/v0.1/whiteboard


Main Program Flow
Start Main
Initialize Load Balancer
Start Monitoring Thread
Generate Tasks
Submit Tasks
Wait for Completion
Display Statistics
End

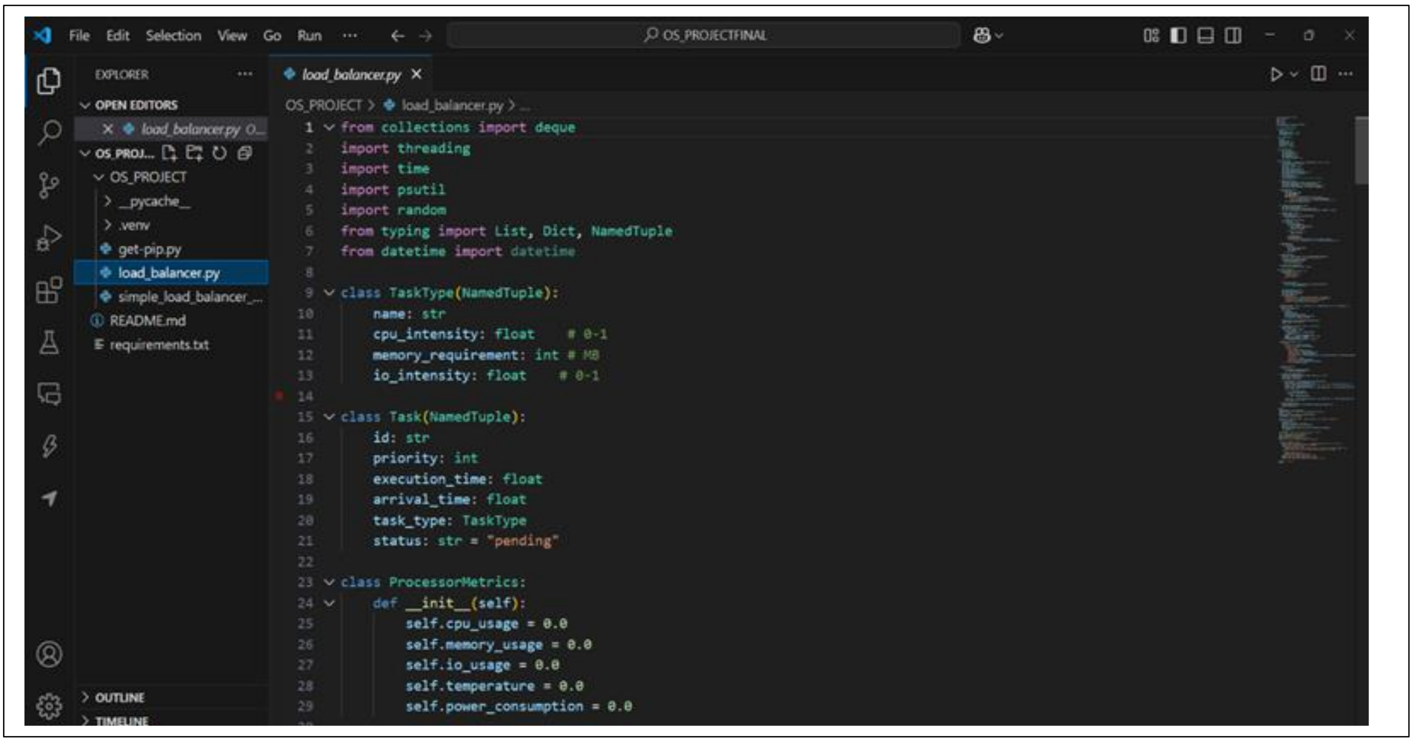

ProcessorQueue Module
Initialize Queue
Add Task
Update Metrics
Simulate Execution
Update Load
Remove Task
Update Statistics
Return

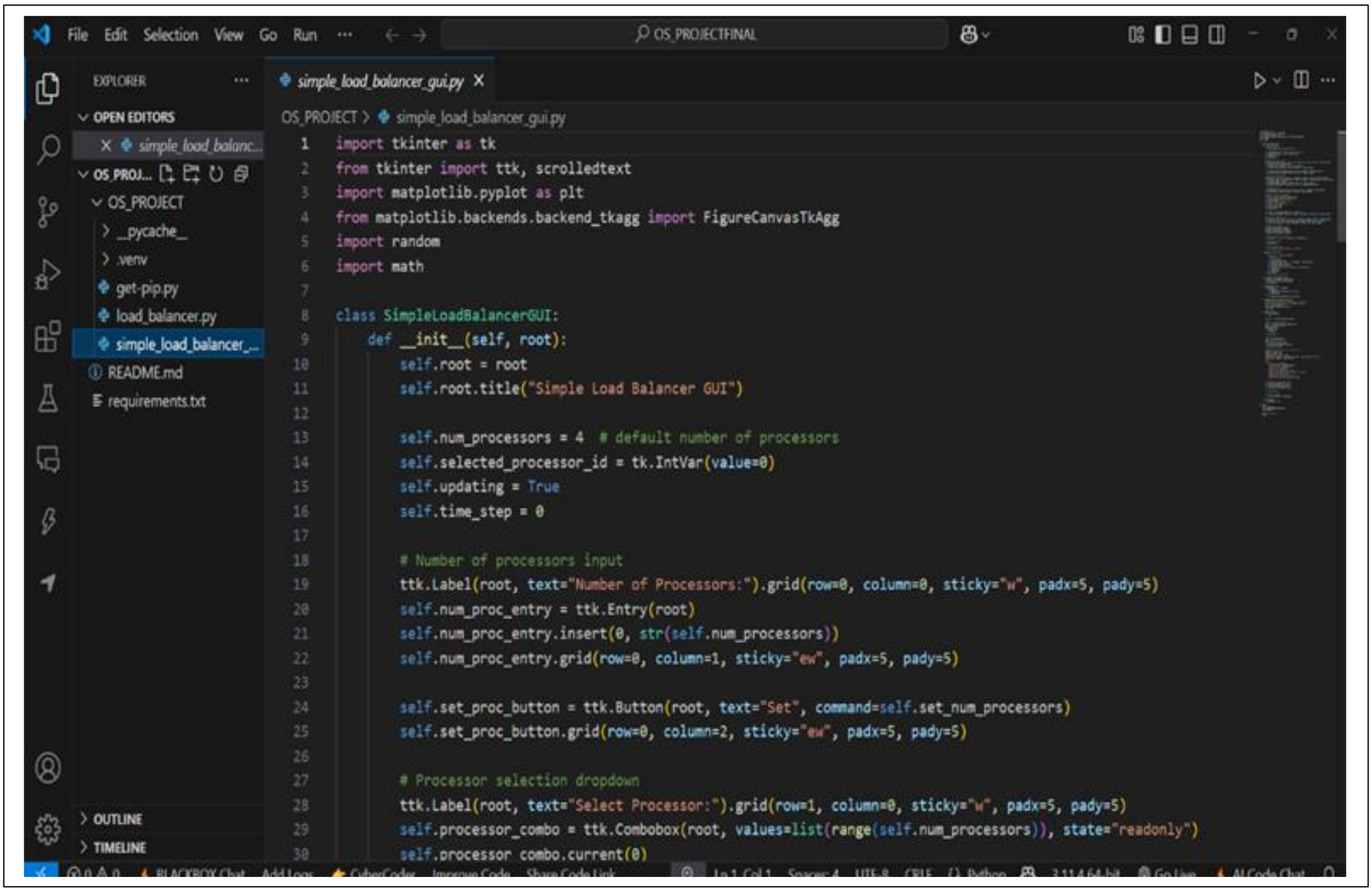

DynamicLoadBalancer Module
Initialize Balancer
Submit Task
Find Optimal Processor
Add to Processor Queue
Monitor Load
Balance Load
Redistribute Tasks
Collect Statistics
Return


SimpleLoadBalancerGUI
Initialize GUI
Set Number of Processors
Select Processor
Generate Artificial Load
Update Load Chart
Update Statistics Text
Handle Pause/Resume
Schedule Next Update
Handle Quit
Return

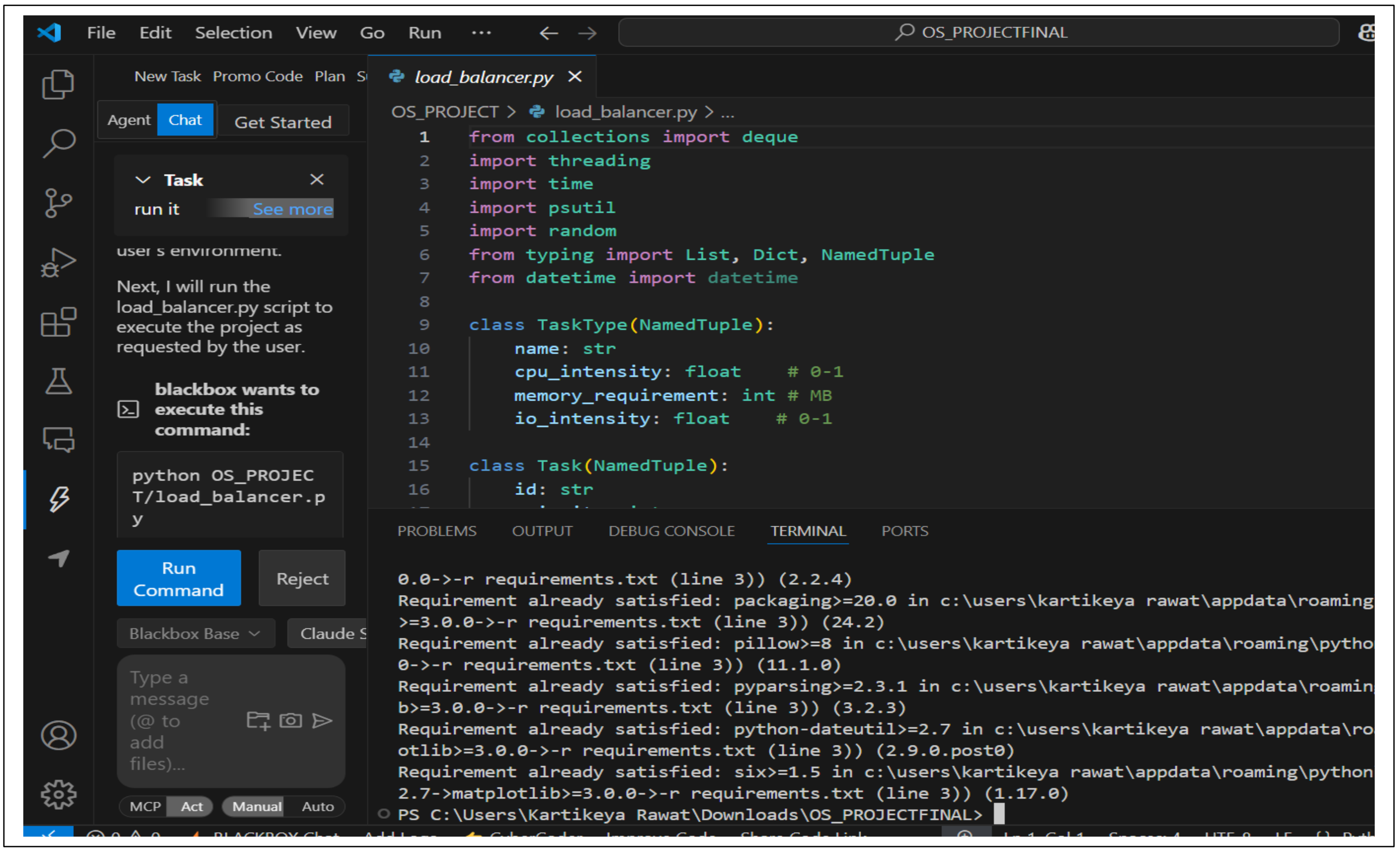
Screenshots

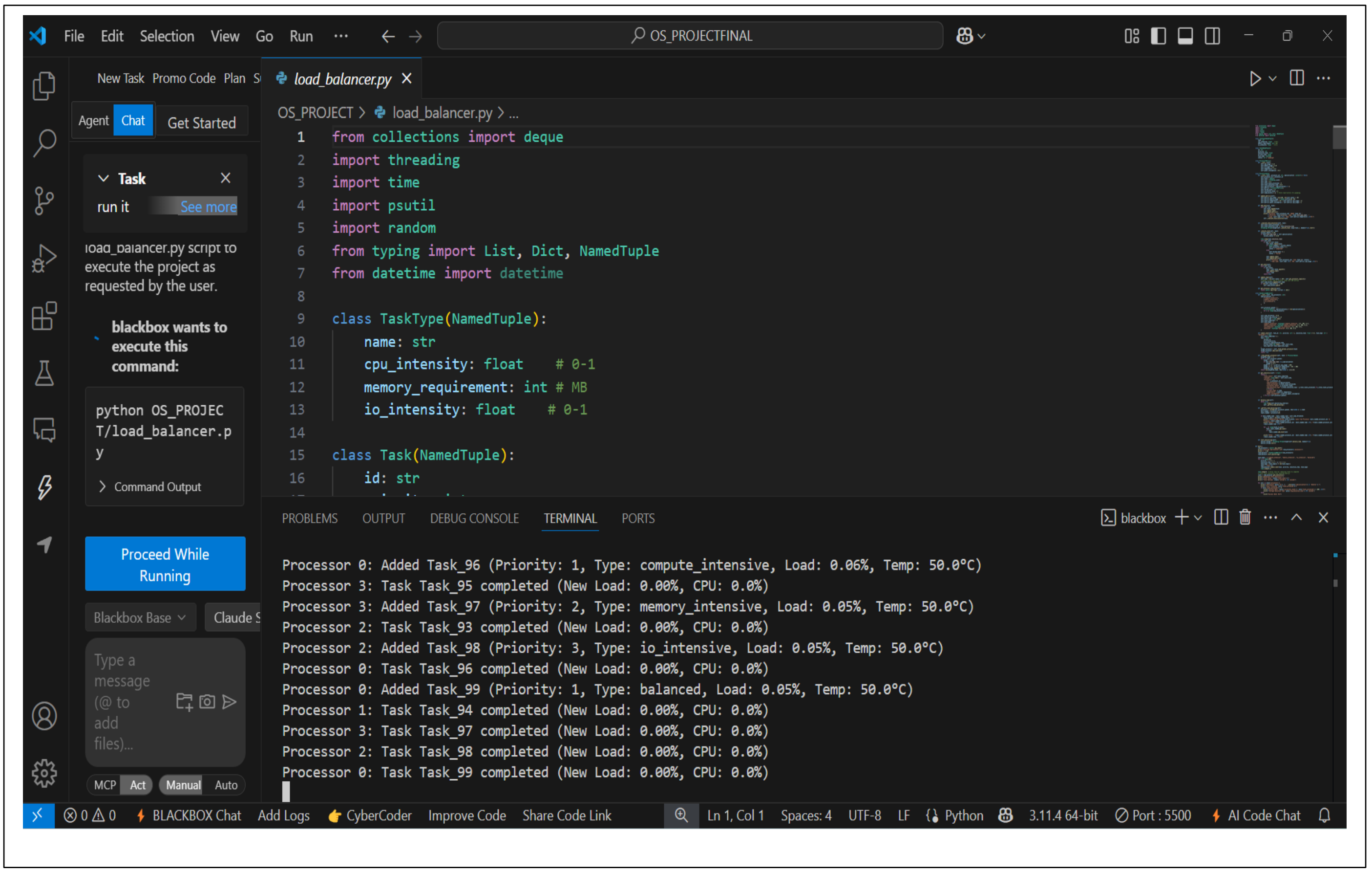
load_balancer.py



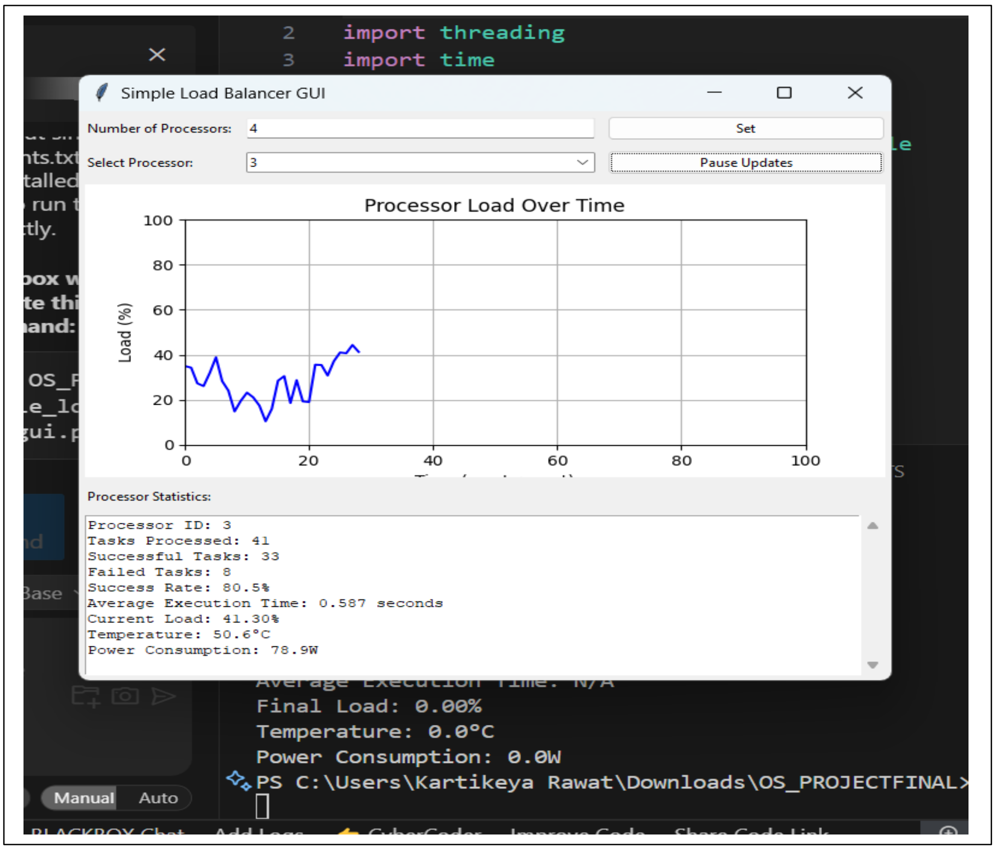


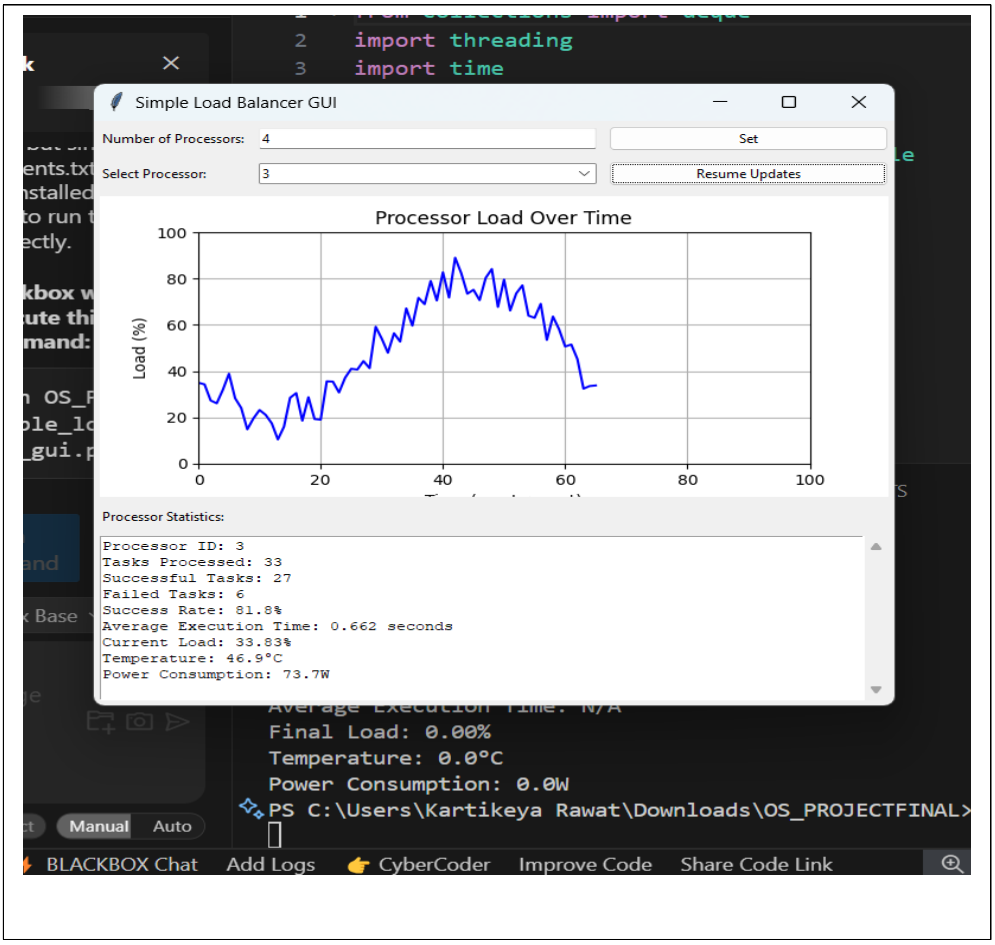
Running





GUI Screenshot





Revision Tracking on GitHub

• Repository Name: DYNAMIC_LOAD_BALANCER

• GitHub Link: https://github.com/KARTIKEYARAWAT/DYNAMIC_LOAD_BALANCER

Conclusion & Future Scope

This dynamic load balancer is like a smart traffic cop for your computer's tasks, directing them to the right processor at the right time. It keeps an eye on everything, making sure no processor is sweating too hard or getting too hot. Watching it in action—seeing tasks get handed out, finished, or even messed up occasionally—gives a real glimpse into how computers juggle so much at once. It's been a mix of getting stuff done and learning how computers manage their resources.

There's a ton of potential to take this project further! Picture adding a mobile app to check on your computer's performance from anywhere. Or using machine learning to guess when tasks will pile up and get ready for them. We could even stretch it to handle tasks across multiple computers in the cloud or make the dashboard more eye-catching with better visuals. The possibilities are huge, and it's exciting to think about how this project could evolve into something even more awesome, helping both tech fans and pros manage their systems like a boss.

What's Next?

• Mobile app to monitor processor loads on my phone
• Add AI to predict busy periods and prep resources
• Cloud version for distributed systems
• Sleek dashboard with live performance graphs

References

- psutil Documentation
- matplotlib Documentation
- tkinter Documentation

YOUTUBE CHANNELS

ByteByteGo https://youtu.be/dBmxNsS3BGE?si=zEWcXbBdZx5_yBxq

5 Minutes Engineering https://youtu.be/TO3yGao_Vjs?si=9sFlOzEXzYnJMO4Q

Tec2Check https://youtu.be/SdzAOanwrF0?si=CLlAIAoMiuhinfeI

Anton Putra https://youtu.be/gqb7LmmXuyw?si=364NdMnfvIXwf4Hl

Concept && Coding – by Shrayansh https://youtu.be/vJYycNWAYZU?si=O-X7JGnBOzeqd__O

Zeto https://youtu.be/c2zhbf1gZHc?si=b_Jtpxbp6n-AWtkV
Zeto https://youtu.be/XFeI_5RIAuY?si=PrBIm1nb3WKta5cu

Appendix

A. AI-Generated Project Elaboration/Breakdown Report

The is a dynamic load balancer designed to optimize task distribution across multiprocessor systems. It intelligently assigns tasks to processors based on their current workload, temperature, and power consumption, ensuring optimal performance and resource utilization. The system includes real-time monitoring and automatic load balancing to prevent any single processor from becoming overwhelmed.

Key Components:

1. **Processor Queue Management**: Manages task queues for each processor, ensuring tasks are executed in order and preventing overload.
2. **Dynamic Load Balancer**: Distributes tasks based on processor workload and specialization, redistributing tasks as needed.
3. **Task Submission and Management**: Submits tasks to the load balancer and tracks their status.
4. **Real-time Monitoring and Visualization**: Provides a GUI to visualize processor load and display detailed statistics.

Technologies Used:

- **Python 3.7+**: Primary programming language.
- **psutil**: For system and process monitoring.
- **matplotlib**: For data visualization.
- **GitHub**: For version control and collaboration.

B. Problem Statement

In multiprocessor systems, inefficient task distribution often leads to some processors being overloaded while others remain underutilized. This results in wasted resources, increased latency, and reduced system performance. The challenge is to dynamically balance the workload across processors to ensure optimal resource utilization and system efficiency.

C. Solution/Code

load_balancer.py

```
from collections import deque

import threading

import time

import psutil

import random

from typing import List, Dict, NamedTuple

from datetime import datetime


class TaskType(NamedTuple):

    name: str

    cpu_intensity: float    # 0-1

    memory_requirement: int # MB

    io_intensity: float    # 0-1


class Task(NamedTuple):

    id: str

    priority: int

    execution_time: float

    arrival_time: float

    task_type: TaskType

    status: str = "pending"


class ProcessorMetrics:
```

```python
    def __init__(self):
        self.cpu_usage = 0.0
        self.memory_usage = 0.0
        self.io_usage = 0.0
        self.temperature = 0.0
        self.power_consumption = 0.0

class ProcessorQueue:
    def __init__(self, processor_id: int, specialization: List[str] = None):
        self.processor_id = processor_id
        self.tasks = deque()
        self.lock = threading.Lock()
        self.load = 0.0
        self.total_tasks_processed = 0
        self.total_execution_time = 0.0
        self.specialization = specialization or []
        self.metrics = ProcessorMetrics()
        self.failed_tasks = 0
        self.successful_tasks = 0
        self.load_history = []  # Store load history for graphing

    def update_metrics(self):
        self.metrics.cpu_usage = min(100, len(self.tasks) * 20)
        self.metrics.memory_usage = random.uniform(20, 80)
        self.metrics.temperature = 40 + (self.metrics.cpu_usage / 2)
        self.metrics.power_consumption = self.metrics.cpu_usage * 2

    def add_task(self, task):
        with self.lock:
```

```
        self.tasks.append(task)
        self.update_load()
        self.update_metrics()
        print(f"Processor {self.processor_id}: Added {task.id} "
              f"(Priority: {task.priority}, Type: {task.task_type.name}, "
              f"Load: {self.load:.2f}%, Temp: {self.metrics.temperature:.1f}°C)")
        self._simulate_task_execution(task)


    def _simulate_task_execution(self, task):
        self.total_tasks_processed += 1
        self.total_execution_time += task.execution_time
        threading.Thread(target=self._execute_task, args=(task,), daemon=True).start()

    def _execute_task(self, task):
        success_chance = 0.95
        if task.task_type.name in self.specialization:
            success_chance += 0.05

        time.sleep(task.execution_time)
        with self.lock:
            if task in self.tasks:
                self.tasks.remove(task)
                if random.random() < success_chance:
                    self.successful_tasks += 1
                    status = "completed"
                else:
                    self.failed_tasks += 1
                    status = "failed"
```

```python
            self.update_load()
            self.update_metrics()
            print(f"Processor {self.processor_id}: Task {task.id} {status} "
                f"(New Load: {self.load:.2f}%, CPU: {self.metrics.cpu_usage:.1f}%)")

    def get_task(self):
        with self.lock:
            if self.tasks:
                task = self.tasks.popleft()
                self.update_load()
                return task
            return None

    def update_load(self):
        self.load = len(self.tasks) * 100 / self.get_processor_capacity()
        # Append current load to history, keep last 100 entries
        self.load_history.append(self.load)
        if len(self.load_history) > 100:
            self.load_history.pop(0)

    def get_processor_capacity(self):
        return psutil.cpu_freq().current or 100.0

class DynamicLoadBalancer:
    def __init__(self, num_processors: int):
        specializations = [
            ["compute_intensive"],
            ["memory_intensive"],
```

```
            ["io_intensive"],
            []
        ]

        self.processor_queues = [
            ProcessorQueue(i, specializations[i % len(specializations)])
            for i in range(num_processors)
        ]

        self.load_threshold = 70.0
        self.monitoring_interval = 1.0
        self.start_time = time.time()
        self.tasks_submitted = 0
        self.task_types = {
            "compute_intensive": TaskType("compute_intensive", 0.9, 200, 0.1),
            "memory_intensive": TaskType("memory_intensive", 0.3, 800, 0.2),
            "io_intensive": TaskType("io_intensive", 0.2, 100, 0.9),
            "balanced": TaskType("balanced", 0.5, 400, 0.5)
        }

    def submit_task(self, task_id: str, priority: int = 1, execution_time: float = 0.5, task_type: str = "balanced") -> bool:
        self.tasks_submitted += 1
        task = Task(
            id=task_id,
            priority=priority,
            execution_time=execution_time,
            arrival_time=time.time() - self.start_time,
            task_type=self.task_types[task_type]
        )
```

```
        target_processor = self._find_optimal_processor(task)
        target_processor.add_task(task)
        return True

    def _find_optimal_processor(self, task) -> ProcessorQueue:
        weighted_loads = []
        for p in self.processor_queues:
            weight = p.load
            if task.task_type.name in p.specialization:
                weight *= 0.7
            weight *= (1 + p.metrics.cpu_usage / 200)
            weight *= (1 + (p.metrics.temperature - 40) / 100)
            weighted_loads.append((p, weight))
        return min(weighted_loads, key=lambda x: x[1])[0]

    def get_statistics(self) -> Dict:
        return {
            "total_tasks": self.tasks_submitted,
            "runtime": time.time() - self.start_time,
            "processors": [{
                "id": p.processor_id,
                "specialization": p.specialization,
                "tasks_processed": p.total_tasks_processed,
                "successful_tasks": p.successful_tasks,
                "avg_execution_time": p.total_execution_time / p.total_tasks_processed if
p.total_tasks_processed > 0 else 0,
                "current_load": p.load,
                "temperature": p.metrics.temperature,
                "power_consumption": p.metrics.power_consumption
            } for p in self.processor_queues]
```

```
        }

    def balance_load(self):
        while True:
            time.sleep(self.monitoring_interval)
            self._perform_load_balancing()

    def _perform_load_balancing(self):
        processors = sorted(self.processor_queues, key=lambda x: x.load)
        most_loaded = processors[-1]
        least_loaded = processors[0]

        if most_loaded.load - least_loaded.load > self.load_threshold:
            tasks_to_move = len(most_loaded.tasks) // 2
            print(f"\nRebalancing: Moving {tasks_to_move} tasks from Processor
{most_loaded.processor_id} to Processor {least_loaded.processor_id}")
            print(f"Before - P{most_loaded.processor_id}: {most_loaded.load:.2f}%,
P{least_loaded.processor_id}: {least_loaded.load:.2f}%")

            for _ in range(tasks_to_move):
                task = most_loaded.get_task()
                if task:
                    least_loaded.add_task(task)

            print(f"After  - P{most_loaded.processor_id}: {most_loaded.load:.2f}%,
P{least_loaded.processor_id}: {least_loaded.load:.2f}%\n")

    def start_monitoring(self):
        monitor_thread = threading.Thread(target=self.balance_load, daemon=True)
        monitor_thread.start()
```

```
def main():
    num_processors = psutil.cpu_count()
    print(f"Starting load balancer with {num_processors} processors")
    print("=" * 50)
    load_balancer = DynamicLoadBalancer(num_processors)
    load_balancer.start_monitoring()

    task_types = ["compute_intensive", "memory_intensive", "io_intensive", "balanced"]
    for i in range(100):
        priority = i % 3 + 1
        execution_time = 0.2 + (i % 5) * 0.1
        task_type = task_types[i % len(task_types)]
        task = f"Task_{i}"
        load_balancer.submit_task(task, priority, execution_time, task_type)
        time.sleep(0.1)

    time.sleep(5)  # Allow time for remaining tasks to complete
    # Print final statistics with enhanced metrics
    stats = load_balancer.get_statistics()
    print("\nEnhanced Final Statistics:")
    print(f"Total Tasks: {stats['total_tasks']}")
    print(f"Total Runtime: {stats['runtime']:.2f} seconds")

    for proc in stats['processors']:
        print(f"\nProcessor {proc['id']} ({', '.join(proc['specialization']) or 'General'}):")
        print(f"Tasks Processed: {proc['tasks_processed']}")
        if proc['tasks_processed'] > 0:
            print(f"Success Rate: {(proc['successful_tasks'] / proc['tasks_processed'] *
100):.1f}%")
```

```
            print(f"Average Execution Time: {proc['avg_execution_time']:.3f} seconds")
        else:
            print("Success Rate: N/A")
            print("Average Execution Time: N/A")
        print(f"Final Load: {proc['current_load']:.2f}%")
        print(f"Temperature: {proc['temperature']:.1f}°C")
        print(f"Power Consumption: {proc['power_consumption']:.1f}W")

if __name__ == "__main__":
    main()
```

simple_load_balancer_gui.py

```
import tkinter as tk
from tkinter import ttk, scrolledtext
import matplotlib.pyplot as plt
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
import random
import math

class SimpleLoadBalancerGUI:
    def __init__(self, root):
        self.root = root
        self.root.title("Simple Load Balancer GUI")

        self.num_processors = 4  # default number of processors
        self.selected_processor_id = tk.IntVar(value=0)
        self.updating = True
        self.time_step = 0
```

```
        # Number of processors input
        ttk.Label(root, text="Number of Processors:").grid(row=0, column=0, sticky="w",
padx=5, pady=5)
        self.num_proc_entry = ttk.Entry(root)
        self.num_proc_entry.insert(0, str(self.num_processors))
        self.num_proc_entry.grid(row=0, column=1, sticky="ew", padx=5, pady=5)

        self.set_proc_button = ttk.Button(root, text="Set", command=self.set_num_processors)
        self.set_proc_button.grid(row=0, column=2, sticky="ew", padx=5, pady=5)

        # Processor selection dropdown
        ttk.Label(root, text="Select Processor:").grid(row=1, column=0, sticky="w", padx=5,
pady=5)
        self.processor_combo = ttk.Combobox(root, values=list(range(self.num_processors)),
state="readonly")
        self.processor_combo.current(0)
        self.processor_combo.grid(row=1, column=1, sticky="ew", padx=5, pady=5)
        self.processor_combo.bind("<<ComboboxSelected>>", self.on_processor_selected)

        # Pause/Resume button
        self.pause_button = ttk.Button(root, text="Pause Updates",
command=self.toggle_updates)
        self.pause_button.grid(row=1, column=2, sticky="ew", padx=5, pady=5)

        # Matplotlib figure for load graph
        self.fig, self.ax = plt.subplots(figsize=(6, 3))
        self.ax.set_title("Processor Load Over Time")
        self.ax.set_xlabel("Time (most recent)")
        self.ax.set_ylabel("Load (%)")
        self.line, = self.ax.plot([], [], 'b-')
```

```
        self.ax.set_ylim(0, 100)
        self.ax.set_xlim(0, 100)
        self.ax.grid(True)

        self.canvas = FigureCanvasTkAgg(self.fig, master=root)
        self.canvas.get_tk_widget().grid(row=2, column=0, columnspan=3, sticky="nsew",
padx=5, pady=5)

        # Text box for processor stats
        ttk.Label(root, text="Processor Statistics:").grid(row=3, column=0, sticky="w", padx=5,
pady=5)
        self.stats_text = scrolledtext.ScrolledText(root, width=80, height=10, state='disabled')
        self.stats_text.grid(row=4, column=0, columnspan=3, sticky="nsew", padx=5, pady=5)

        # Configure grid weights
        root.grid_rowconfigure(2, weight=1)
        root.grid_rowconfigure(4, weight=1)
        root.grid_columnconfigure(1, weight=1)
        root.grid_columnconfigure(2, weight=1)

        # Initialize artificial load data
        self.load_data = {i: [] for i in range(self.num_processors)}

        # Start periodic update
        self.update_gui()

        # Bind close window event
        self.root.protocol("WM_DELETE_WINDOW", self.on_quit)

    def set_num_processors(self):
```

```
    try:
        num = int(self.num_proc_entry.get())
        if num <= 0:
            raise ValueError
        self.num_processors = num
        self.processor_combo['values'] = list(range(self.num_processors))
        self.processor_combo.current(0)
        self.selected_processor_id.set(0)
        self.load_data = {i: [] for i in range(self.num_processors)}
        self.time_step = 0
        self.update_gui()
    except ValueError:
        pass  # ignore invalid input

def on_processor_selected(self, event):
    selected = self.processor_combo.current()
    self.selected_processor_id.set(selected)
    self.update_gui()

def toggle_updates(self):
    self.updating = not self.updating
    if self.updating:
        self.pause_button.config(text="Pause Updates")
        self.update_gui()
    else:
        self.pause_button.config(text="Resume Updates")

def generate_artificial_load(self, proc_id):
    # Generate artificial load data using sine wave + random noise
```

```
        base = 50 + 30 * math.sin(0.1 * self.time_step + proc_id)
        noise = random.uniform(-10, 10)
        load = max(0, min(100, base + noise))
        return load

    def update_gui(self):
        if not self.updating:
            return

        proc_id = self.selected_processor_id.get()

        # Update artificial load data
        load = self.generate_artificial_load(proc_id)
        data = self.load_data[proc_id]
        data.append(load)
        if len(data) > 100:
            data.pop(0)

        self.time_step += 1

        # Update load line chart
        xdata = list(range(len(data)))
        ydata = data
        self.line.set_data(xdata, ydata)
        self.ax.set_xlim(0, max(100, len(data)))
        self.ax.set_ylim(0, 100)

        # Update processor stats text with artificial data
        success = int(load * 0.8)
```

```python
        failed = int(load * 0.2)
        tasks_processed = success + failed
        success_rate = (success / tasks_processed * 100) if tasks_processed > 0 else 0
        avg_exec_time = max(0.1, 1.0 - load / 100)

        text = (
            f"Processor ID: {proc_id}\n"
            f"Tasks Processed: {tasks_processed}\n"
            f"Successful Tasks: {success}\n"
            f"Failed Tasks: {failed}\n"
            f"Success Rate: {success_rate:.1f}%\n"
            f"Average Execution Time: {avg_exec_time:.3f} seconds\n"
            f"Current Load: {load:.2f}%\n"
            f"Temperature: {30 + load * 0.5:.1f}°C\n"
            f"Power Consumption: {50 + load * 0.7:.1f}W\n"
        )

        self.stats_text.config(state='normal')
        self.stats_text.delete(1.0, tk.END)
        self.stats_text.insert(tk.END, text)
        self.stats_text.config(state='disabled')

        self.canvas.draw()

        # Schedule next update
        self.root.after(1000, self.update_gui)

    def on_quit(self):
        self.updating = False
```

```
def main():
    root = tk.Tk()
    app = SimpleLoadBalancerGUI(root)
    root.mainloop()


if __name__ == "__main__":
    main()
```

requirement.txt

```
psutil>=5.9.0
typing>=3.7.4
matplotlib>=3.0.0
```