Distributed OS Resource Monitoring System

Members:

Naveen Bollaboyina

Narendra Papasani

1. Introduction & Problem Statement

The Problem: System administrators in universities and offices manage 100s of PCs but lack proper tools.

The "Complexity Gap"

Enterprise Tools (e.g., Zabbix)

Too complex to set up and maintain.

Cloud-Native Tools (e.g., Prometheus)

Not designed for static fleets of PCs and use a "pull" model that is difficult with firewalls.

Manual Checks (e.g., SSH)

Not scalable and provide no historical data.

Our Goal

To build a system that is:

- **Easy to Deploy**
- **Simple to Maintain**

Scalable to 100s of agents

Provides **Deep, Actionable Insights** (not just data)

2. Literature Survey

We analyzed existing open-source solutions to identify a market gap.

Tool	Strength	Weakness for Our Use Case
Zabbix	Extremely powerful, many features.	Very complex to install, configure, and maintain.
Prometheus	Excellent for cloud (Kubernetes).	"Pull" model is difficult for PCs behind firewalls.
Netdata	Beautiful real-time, per-PC data.	Centralized, multi-PC dashboard is a paid cloud service.

Our Niche

We found a clear need for a **self-hosted**, **push-based**, and **easy-to-deploy** system with a modern, multi-tenant UI.

3. Our Proposed Solution: A 3-Tier System

The Agent (Python)

A lightweight, cross-platform script that runs on each client PC. Intelligently "pushes" data to the server.

The Server (Docker)

A high-performance, containerized backend that receives and processes all data.

The Dashboard (React)

A secure, multi-tenant web application for admins to view live data, manage PCs, and get insights.

4. System Architecture

Data Flow

1

Agent Pushes Data

The Agent **pushes** JSON data via HTTPS to the FastAPI Server.

2

API Validates & Queues

The API validates the key and instantly places the job onto the **RabbitMQ Queue** (this is very fast).

3

Worker Processes

A separate **Python Worker** (the "Chef") pulls from the queue at a steady pace.

4

Data Storage

The Worker processes the data and writes it to the **TimescaleDB Database**.

Key Insight

This decoupled design means the API never gets stuck. It can handle a of 100s of agents at once without crashing the database.

5. Implementation: The Agent

Cross-Platform

Built in Python using the psutil library. It runs on Windows, Linux, and macOS.

Resilient

Offline Buffering: If the server is down, the agent saves data in a local queue and sends it when the server is back. **No data is lost.**

"Thundering Herd" Protection: A random startup "jitter" prevents all agents from hitting the server at the same time after a power outage.

Lightweight

Minimal CPU and RAM footprint (as shown in our results).

6. Dynamic Thresholds & Deep Analysis

This is our "smart" feature that provides actionable insights.

1

Normal Mode (CPU < 85%)

Interval: 10 seconds

Payload: Basic metrics (CPU, RAM, Network I/O)

2

Alert Mode (CPU > 85%)

The agent automatically speeds up. Interval becomes **5 seconds**.

The agent performs **Deep Analysis**: It collects the **Top 5 Processes** (by CPU/RAM) and sends them with the metrics.

Result

The admin doesn't just see "CPU is 90%." They see, "CPU is 90% *because* chrome.exe is stuck at 72%."

7. Implementation: The Server

FastAPI

A high-performance Python framework for the API endpoint.

RabbitMQ

The message broker. It acts as a "shock absorber" for data spikes and ensures resilience.

TimescaleDB (PostgreSQL)

A specialized database for time-series data. We use "hypertables" for extremely fast queries, even with billions of rows.

Multi-Tenancy

- The users table stores admin accounts.
- Each user has a unique api_key.
- The agents table is linked via user_id.

This ensures an admin can *only* see the agents that belong to them.

8. "One-Click" Deployment

A core goal was **ease of management**. We achieved this with **Docker**.

The entire server stack (FastAPI, TimescaleDB, RabbitMQ) is defined in a single docker-compose.yml file.

Deployment is Simple

Clone Repository

git clone ...

Deploy with Docker

sudo docker-compose up -d --build

Platform-Agnostic

The *exact same* file and command can be used to deploy on:

A Physical Server

in the college lab.

A Cloud VM

on AWS, Azure, or any provider.

9. Implementation: The Dashboard

Built with **React (Vite)** and **Material-UI (MUI)** for a clean, professional UI.

Secure

Uses **JWT (JSON Web Tokens)** for admin login.

- All API calls are authenticated.

Data-Driven

Main Page: A clear, sortable **Table** of all agents with their live "Online/Offline" status.

Detail Page: Clicking an agent shows all its data.

Visual

Uses **Recharts** to render live-updating line graphs for all metrics.

10. Results: (Agent Performance)

Goal:

To prove the agent is lightweight and won't slow down client PCs.

Method:

Ran the real agent on a host PC and measured its resource usage.

Results:

The agent's resource footprint is negligible.

Agent Mode	Avg. CPU Usage	Avg. Memory Usage
Normal (10s interval)	<0.08%	<20 MB
Alert (5s + Processes)	<0.2%	<20 MB

Observation:

The agent is extremely efficient. Even during a "Deep Analysis" alert, the performance impact is almost zero, making it safe for any PC.

11. Results: (Server Scalability)

Goal:

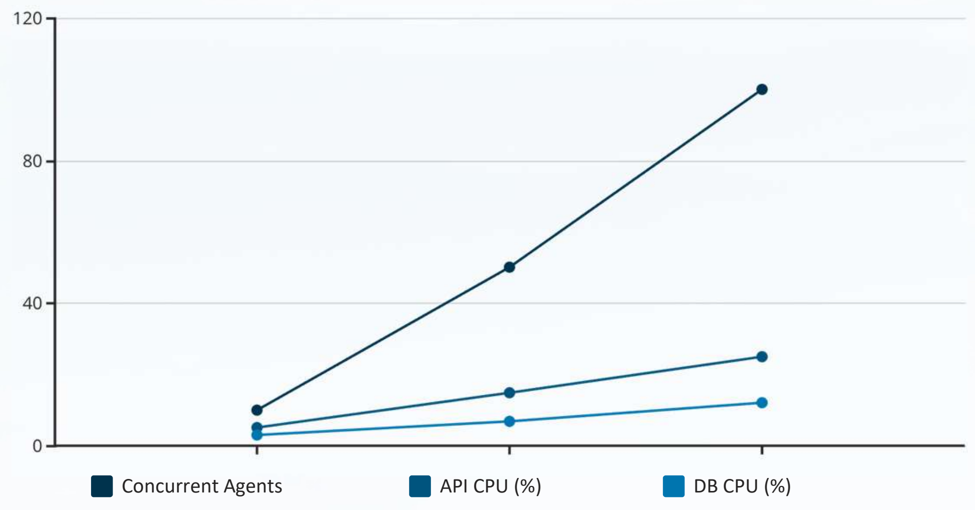
To prove the server's decoupled architecture can handle 100s of agents.

Method:

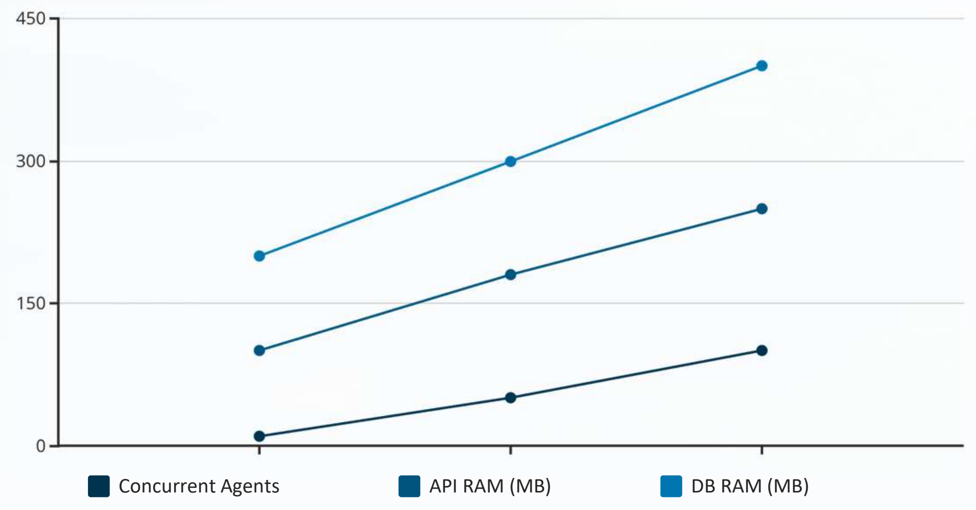
Ran our agent_simulator.py with 10, 50, and 100 concurrent agents, simulating real-world load.

Results:

Server CPU Usage with Increasing Agents



Server RAM Usage with Increasing Agents



Both CPU and RAM usage for the API and Database components show a stable, linear increase, indicating excellent scalability and resource management even with 100 active agents.

12. Observations & Discussion

1

Message Queue as Buffer

The database's CPU and RAM usage remained consistently low, even under simulated high loads. This demonstrates RabbitMQ's exceptional capability to effectively buffer incoming data, shielding the database from "thundering herd" scenarios.

2

Linear Scalability

The API container exhibited a linear increase in resource utilization, which is expected behavior under increasing load. Crucially, the overall system maintained stability and responsiveness, confirming its scalable design.

3

High Maintainability

Even with 100 concurrent agents generating over 600 metrics per minute, the server remained healthy and the RabbitMQ queue stayed empty. This validates the worker's efficiency and the system's inherent maintainability under significant operational demands.

13. Conclusion & Future Work

Key Achievements

Problem Solved

A scalable, multi-tenant monitoring system addressing the "complexity gap" in distributed resource management.

Core Strengths

Features easy deployment (Docker), scalable architecture (RabbitMQ), and deep insights (Dynamic Thresholds).

Validated Performance

Proven lightweight on client devices and robust, resilient on the server side under load.

Next Steps

Remote Agent Deployment

Develop a dashboard feature to remotely push and install agents on new client machines, streamlining setup.

Automated Alerting

Integrate external services for email/SMS notifications, allowing admins to receive critical alerts without constant dashboard monitoring.

Thank You!

Questions & Answers

Please feel free to ask anything about our project, from concept to implementation.