DEVELOPMENT OF AN AI DRIVEN SYSTEM MONITOR AND PROCESS MANAGER FOR WINDOWS OS

BY

OSEGHALE JENNIFER EBEHIREMHEN

ENG2002509

IN PARTIAL FUFILMENT OF THE REQUIREMENT FOR THE AWARD OF BACHELOR OF ENGINEERING DEGREE

DEPARTMENT OF COMPUTER ENGINEERING

FACULTY OF ENGINEERING

UNIVERSITY OF BENIN

PROJECT SUPERVISOR: DR. OMOIFO

OCTOBER 2025

DEVELOPMENT OF AN AI DRIVEN SYSTEM MONITOR AND PROCESS MANAGER FOR WINDOWS OS

BY

OSEGHALE JENNIFER EBEHIREMHEN

ENG2002509

IN PARTIAL FUFILMENT OF THE REQUIREMENT FOR THE AWARD OF BACHELOR OF ENGINEERING DEGREE

DEPARTMENT OF COMPUTER ENGINEERING

FACULTY OF ENGINEERING

UNIVERSITY OF BENIN

PROJECT SUPERVISOR: DR. OMOIFO

OCTOBER 2025

CERTIFICATION

I certify that this project **Development of an AI Driven System Monitor and Process Manager for Windows OS** was carried out by **Oseghale Jennifer Ebehiremhen, ENG2002509**, in the **Department of Computer Engineering, Faculty of Engineering, University of Benin, Benin City** and is accepted for proposal.

_________________________ _____________

Engr. Dr. I.A. Edeoghon **Date**
Head of Department

_________________________ ______________

Dr. Omoifo **Date**
Project Supervisor

DEDICATION

This entire project is dedicated, first and foremost, to Almighty God. It is through His divine grace, wisdom, and sustaining strength that I found the direction and endurance necessary to complete this academic challenge. His presence was my persistent source of inspiration and determination.

I also dedicate this achievement to my dearest family. Their unyielding love, crucial support, and constant encouragement served as an indispensable foundation for my efforts. Their personal sacrifices and consistent expression of confidence in me were a continuous catalyst for reaching my objectives.

This document is a direct acknowledgment of the commitment, patience, and invaluable mentorship provided by those who stood with me. I hold their support in everlasting esteem.

ACKNOWLEDGEMENTS

I first and foremost appreciate God, my source of wisdom and guidance, who has brought me through and been a constant comforter and source of inspiration all through this undertaking.

I extend my profound appreciation to my esteemed project supervisor, Dr Omoifo, for his invaluable supervision, patience, and mentorship during the course of this project. Your insightful feedback truly pushed this work to a new level.

I wholeheartedly appreciate my parents, Mr and Mrs Emmanuel Oseghale, for their unwavering love, support, and sacrifices made to ensure my comfort and success throughout my academic journey.

I sincerely appreciate my mentor, Mordecai Eletukudo for his immense support and aid provided throughout this journey. I also appreciate my wonderful educational mother, Dr Mrs Okodun, for her endless support and advise throughout my academic journey, my course advisor, Mr Sylvester Akinbohun and my ever-supportive Head of Department, Engr Dr Edeoghon. God bless you all

Lastly, a special mention goes to my friends, Sharon, Jennifer, Laura, Sobechi, Maro for having my back all through. Your love and encouragement have been unwavering, pushing me to strive for excellence in all that I do.

TABLE OF CONTENTS

CERTIFICATION ii
DEDICATION iii
ACKNOWLEDGEMENTS iv
TABLE OF CONTENTS v
ABSTRACT viii
CHAPTER 1 1
1. INTRODUCTION 1
1.1 BACKGROUND OF THE STUDY 1
1.2 PROBLEM STATEMENT 2
1.3 AIMS AND OBJECTIVES 3
1.4 SCOPE OF STUDY 3
1.5 RELEVANCE OF THE STUDY 4
CHAPTER 2 5
2. LITERATURE REVIEW 5
2.1 OVERVIEW OF AI IN SYSTEM MONITORING 5
2.2 ARTIFICIAL INTELLIGENCE FOR PROCESS AND RESOURCE MANAGEMENT 5
2.3 THE ROLE OF RUST IN BUILDING EFFICIENT AND SAFE SYSTEM TOOLS 6
2.4 WINDOWS OS AS THE TARGET PLATFORM 7
2.5 INTEGRATION OF AI APIS (GEMINI) AND SYSTEM METRICS LIBRARIES (SYSINFO) 8
2.5.1 RATIONALE FOR GEMINI API SELECTION 8
2.6 RELATED WORKS 10
2.7 EXISTING WINDOWS MONITORING TOOLS AND THEIR LIMITATIONS 11
2.8 META-ANALYSIS TABLE 13
CHAPTER 3 15
3. METHODOLOGY 15
3.1 RESEARCH DESIGN AND APPROACH 15
3.2 SYSTEM ARCHITECTURE OVERVIEW 15
3.3 SYSTEM DEVELOPMENT AND IMPLEMENTATION 16
3.3.3 PROCESS MANAGEMENT MODULE (CONCEPTUAL DESIGN) 22
3.3 TOOLS AND TECHNOLOGIES 24
3.5 TESTING AND EVALUATION METHODOLOGY 27

3.5.4 DATA COLLECTION AND ANALYSIS PROCEDURES 33
3.6 ETHICAL CONSIDERATIONS 33
3.7 LIMITATIONS OF THE METHODOLOGY 34
CHAPTER 4 36
4 RESULTS AND DISCUSSION 36
4.1 OVERVIEW 36
4.2 SYSTEM IMPLEMENTATION VERIFICATION 36
4.2.2 SYSTEM OPERATION WORKFLOW 36
4.3 METRIC ACCURACY VALIDATION 37
4.3.2 VALIDATION RESULTS 37
4.4 AI ANALYSIS OUTPUT EVALUATION 38
4.4.1 PROMPT DESIGN 38
4.5 TESTING SCENARIOS AND OBSERVATIONS 41
4.6 DISCUSSION OF FINDINGS 42
4.6.4 COMPARISON WITH TRADITIONAL MONITORING TOOLS 44
4.5.6 PRACTICAL IMPLICATIONS 44
4.5.7 AREAS FOR FUTURE ENHANCEMENT 45
CHAPTER 5 46
5.1 CONCLUSION 46
5.2 RECOMMENDATIONS AND FUTURE WORK 47
REFRENCES 49
APPENDIX 52

Table 2. 1 16

Table 3. 1 18

Table 3. 2 28

Table 4. 1 42

Table 4. 2 42

ABSTRACT

In today's computing landscape, it is all about the proactive and intelligent solutions that extend beyond the reactive nature of a standard system monitoring solution. This project focuses on creating an AI-based System Monitor and Process Manager for Windows OS in the Rust programming language, with real-time intelligent analysis powered by Google's Gemini API.

It overcomes the shortcomings of traditional monitoring solutions, including fixed thresholds, lack of elasticity for changing workloads, and the absence of human-readable insights, by combining Rust's performance and memory safety with cloud-based large language model (LLM) reasoning. Through the use of the sysinfo crate system metrics, such as CPU utilization, memory, disk I/O, and network traffic are collected and packaged into prompts sent to the Gemini 1.5 Flash API, which provides actionable, natural language summaries for each metric that are categorized by anomalies, optimizations, and priorities.

High metric accuracy was shown through validation testing with the Windows Task Manager (1–3% deviation, which is acceptable for real time monitoring). The AI-generated responses have always been relevant to the content, and 75-80% of them contained specific recommendations. The non-technical evaluators and technical evaluators both provided user feedback that showed a great preference for the use of the natural language interface system as opposed to traditional numerical dashboards.

Although complete automation of the process was not achieved for security and complexity reasons, the system provides a good proof of concept for the feasibility of a hybrid architecture between a low-level native monitoring system and an AI-based semantic analysis system. The tool can be deployed as standalone tool without any dependency that requires a Windows executable file, which makes it easy to use for both individual and enterprise users.

This work presents a novel methodology for proactive system performance management, especially important for resource-constrained systems, and serves as a stepping stone for future advancements such as predictive analytics, automated process control, and local LLM inference.

Key components: System monitoring, Windows OS, Rust programming language, Artificial Intelligence, Gemini API, process management, anomaly detection, real-time analysis.

CHAPTER 1

1. INTRODUCTION

1.1 BACKGROUND OF THE STUDY

With the continuous growth of complexity and requirements of computer systems, the requirement for intelligent monitoring and management solution of desktop operating systems, particularly Microsoft Windows, is no longer an option. As the number of multi-processing applications has grown rapidly, along with the complexity of the various services, new technology had to be developed to be capable of observing and analyzing the performance of the system in real time. One such solution can be an AI-based system monitoring and processing manager that applies AI techniques to process monitoring and performance management.

Traditional monitoring systems and process control based on set thresholds and manual intervention have several drawbacks. This approach is not suitable in some situations due to inability to adapt effectively to varying workloads, leading to suboptimal resource utilization and performance. This is particularly true in today's world where many applications vie for CPU, memory and I/O resources. This causes users to experience issues such as computer lag, system crashes or power problems, and hence, it is not hard to find that there is a gap in system management that needs to be enhanced.

The new AI and ML advancements are offering extremely promising answers to the challenges that these systems present. With the use of sophisticated techniques from AI, systems can carefully examine huge amounts of past and current performance information. With such analyses they can make intelligent decisions about what to do, predict future resource shortages, and identify any anomalies that might be indicators of potential problems. This means that computers can respond to changes in their users' behavior, software requirements, and hardware restrictions in a smart way.

As a modern systems programming language, Rust is well suited as an excellent base language for creating this type of advanced tool. Known for its performance, memory safety, and powerful concurrency capabilities, with no garbage collector overhead, Rust is an ideal language for developing robust and reliable monitoring systems. It is its native ability to do low level operations safely that makes it possible to interact directly and safely with the underlying hardware interfaces, the process table and the Windows OS kernel, all critical to a truly effective system monitor. With careful design and a programming language like Rust, this research project is to build the whole ‘AI-powered systems monitor and process manager’ for the Windows operating system. The idea

behind the proposed tool is to merge the potential and stability of Rust, a language that guarantees zero-risk memory safety, with the ability of machine learning to adapt and predict. The vision is to offer a state-of-the-art solution with real-time, intelligent process control and optimized system performance to meet the constantly evolving, ever increasing needs of personal and enterprise computing.

1.2 PROBLEM STATEMENT

The modern computing environment, which includes the myriad of applications that require a large number of resources, background processes that never go away and an ever-present requirement for connectivity, has been a huge strain on personal computers using the Windows operating system. Whether in a Nigerian classroom or office, or in other developing countries, these PCs are clamoring for critical resources. All the resources, memory, disk bandwidth and battery power are fighting it out in this new way of the competition. The approach of the conventional system monitoring tools available in Windows is limited. They typically:

1. Use very rigid and fixed alert thresholds (e.g., trigger a warning when CPU usage is above 80%).
2. Need manual user interaction to stop or re-prioritize resource-consuming processes.
3. Has no ability to foresee or predict spikes in future workload or system-wide patterns.

Traditional tools are therefore failing to cope with today's multi-process, dynamic workloads, for a number of key and ongoing challenges:

1. Resource Congestion and System Thrashing: When multiple applications demanding high resources, such as video conferencing suites, integrated development environments (IDEs), multiple browser windows, and background updaters, all compete for limited CPU and memory resources, it leads to “resource congestion” and “system thrashing.” This will often cause excessive levels of context switching, paging and overall system performance degradation.

2. Inefficient Resource Utilization: When there is relative inactivity, significant underutilization of compute, memory and I/O capacity. Most important, the default operating system scheduler is not designed to be able to down-clock components or smartly redistribute them to energy-saving tasks such as background compilation or predictive caching.

3. Limited Adaptability and Learning: Static monitoring rules are not able to adapt for unforeseen and sudden changes in workload patterns, such as starting a new antivirus scan

or running a real-time data-science notebook. Moreover, they do not learn from history – from the daily resource bursts during on-line lectures to the nightly software builds – and thus miss many chances to be smarter about resource allocation, throttling, or management.

4. Excessive Power Consumption: Frequently, unchecked processes run in the background and suboptimal scheduling mechanisms keep CPUs in higher clock states and SSDs in constant write cycles. Unchecked processes running in the background and suboptimal scheduling mechanisms often keep CPUs in higher clock states and SSDs in constant write cycles. This directly impacts faster laptop battery usage and higher electric power consumption, especially in areas with unreliable power supplies.

5. Degraded System Reliability and User Experience: A prolonged period of overloaded systems results in regular application freezes, unpredicted operating system crashes (even "blue screen" crashes) and a noticeable decrease in overall user productivity and satisfaction.

1.3 AIMS AND OBJECTIVES

AIM

The major aim of the project is to 'develop an AI driven system monitor and process manager for windows operating system using RUST, to analyses system performance and manage processes in real time.

OBJECTIVES

1. To create an effective information collection process, including timely collection of information, and to monitor the status of systems, including gathering accurate information on important metrics such as CPU utilization, memory usage, disk I/O, network activity etc.
2. To create an AI-driven analysis pipeline to interpret the metrics gathered to detect anomalies, forecast future resource patterns, and give context to understanding the behaviour of a system, using the ChatGPT API.
3. Develop and implement a process manager that will be dynamic, in order to optimize the system performance and resource allocation on the basis of the insights and recommendations that the AI has generated.

1.4 SCOPE OF STUDY

This study will concentrate on developing an AI based system monitor and process manager for Windows using the Rust programming language. The aim is to design a tool to analyze the

performance of system in real time and to intelligently manage the processes to increase the system efficiency. My research area covers:

1. Examine and discuss current Windows system monitoring options including their strengths, weaknesses and potential enhancements.
2. Implementing a monitoring system to keep track of key performance indicators such as CPU usage, memory utilization, disk activity, network performance, etc.
3. Embedding machine learning algorithms which can learn from system behaviour patterns and make automated decisions on process management and optimization.
4. Experimental and simulation testing of the developed system to determine its ability to enhance system speed, stability and resource utilization in comparison to the existing solutions.

1.5 RELEVANCE OF THE STUDY

Today's Windows systems have a higher level of complexity and existing system monitoring solutions are unable to meet modern computing needs. These are the types of tools that are not able to respond to real-time changes quickly enough when running resource-intensive programs or more than one program at a time. This results in issues such as poor performance, inefficiency in the system, and degrades the user experience. In this study, the objective is to overcome this deficiency and create a smarter and automated solution that is able to analyze the performance of the system as it happens, and adjust accordingly without manual assistance. The system is developed using an artificial intelligence approach for monitoring and decision making, and is programmed in Rust, which is known for safety and performance, two characteristics that are crucial for system-level software. This project could be valuable in a personal computer and also in a business setting where the reliability of performance would be important. The developed system is mainly aimed at maintaining the system stability, improvement of performance and effective management of energy consumption. In Nigeria where the efficient use of computer resources and smart power management are important issues, this research is of particular relevance. In these places, many users have to cope with an intermittent power supply and want their systems to run as efficiently as possible. This paper aims to provide a contribution to make computing more reliable and sustainable in harsh environments, by developing a tool to automatically optimize its use of system resources. The aim of this is to highlight the power of integrating AI with the latest programming languages to produce improved solutions to everyday computing problem

CHAPTER 2

2. LITERATURE REVIEW

2.1 OVERVIEW OF AI IN SYSTEM MONITORING

Traditionally, the system monitoring process entails gathering metrics such as CPU consumption, memory usage, disk I/O, process states and network statistics to enable visibility into the health of the system (Kim et al., 2022). The static threshold-based monitoring methodology fails to deliver timely and accurate insights as computing environments become dynamic (e.g. virtualization, containers, hybrid workloads), particularly in the context of Windows OS (Chatterjee & Das, 2024). The need to identify deviations from changing baselines, to relate multi-metric contexts, and to minimize false alarms (Liu et al., 2024) is thus the motivation for moving from rule-based to intelligent, adaptive systems. With that in mind, monitoring systems powered by AI that can learn about "normal" performance, predict issues, and provide guidance on how to fix them are becoming essential for proactive system management (New Relic, 2024).

This landscape is used to support theoretical arguments for using AI (in this case, Gemini) and system metric collection (in this case, sysinfo crate) in a Windows based monitor/process manager.

2.2 ARTIFICIAL INTELLIGENCE FOR PROCESS AND RESOURCE MANAGEMENT

However, as the tools become more sophisticated and intelligent, resource and process management tools are incorporating machine learning (ML) capabilities including anomaly detection, resource prediction and automated remediation. Past workload states have been mapped to predictable future states by relying on supervised algorithms such as random forests, SVMs and neural networks (Moharam et al., 2025). Clustering methods (k-means, DBSCAN) and autoencoders are examples of unsupervised and hybrid approaches that can be used to detect novel behaviours (Chandola et al., 2009; Moharam et al., 2025). Deep learning models like LSTM networks and autoencoders have demonstrated greater capability to handle the temporal structure of data, as seen in system metrics, and to identify subtle shifts or drifts in the data (Lindemann et al., 2021).

This system monitor and process manager uses the sysinfo Rust crate to gather metrics (CPU on Windows, it displays the data (%, RAM %, disk usage, process list) to the Gemini API to return a summarised metrics text which is used to generate concise bullet-point insights. The

end-to-end flow (metric capture → AI summarization→ actionable output) reflects the advances ML-based paradigm rather than traditional static methods.

2.3 THE ROLE OF RUST IN BUILDING EFFICIENT AND SAFE SYSTEM TOOLS

The memory safety, performance and modern abstractions of Rust have made it a popular choice for systems programming (Bugden & Alahmar, 2022; Jung et al., 2017). No classes of errors like null-pointer dereferences, buffer overflows or data races are possible because of the ownership and borrowing model of the language (important for system-level utilities on Windows). Matsakis and Klock (2014) describe "zero-cost abstractions" as a guarantee that high-level constructs do not have an increased runtime cost when compared to other constructs of the same type that exist at a lower level. Moreover, Rust is performance competitive to C/C++ (Balasubramanian et al., 2017), and supports native compilation, which is beneficial to any tool that needs to run with minimum overhead in a production environment.

Use of Rust for the monitor/process manager means that the binary will be lean, performant and safe which is critical when collecting high-frequency system metrics and management of processes on Windows OS. In addition, the language ecosystem (Cargo, crates) makes cross-compilation possible and therefore packaging of the globally installable .exe. Orchestrating these components reliably, Rust is a natural choice, since the system needs to interact with both the lower-level OS metrics (via native Rust crates such as sysinfo) and the higher level AI APIs (via HTTP requests to Gemini).

2.3.1 TECHNICAL IMPLEMENTATION: SYSINFO CRATE VS PSUTIL

The proposal said psutil (a Python library) would be used to collect the metrics, but the implementation is actually provided by rust sysinfo. There are a number of reasons why this is an important choice:

1. **Native Performance**: sysinfo is a completely Rust based library that allows direct access to Windows APIs, bypassing Python interop and subprocess calls to avoid overhead and failure points.
2. **Type Safety**: Systemic benefits from Rust's type safety, which is guaranteed during compilation and ensures memory safety and avoids runtime errors that are often seen in cross-language FFI.
3. **Cross-Platform Compatibility**: This project is for the Windows OS, however sysinfo

offers a shared API across platforms (Windows, Linux, macOS) so that it can be extended in the future.

4. **Lightweight Dependencies**: Lightweight Dependencies: In contrast to the Python subprocess method, using psutil, sysinfo is compiled into the binary, which means there are no other external run-time dependencies.

This architectural design is an example of the project's approach to developing system tools using Rust's ecosystem to create efficient and compact systems.

2.4 WINDOWS OS AS THE TARGET PLATFORM

Over the last few years, Windows OS has been the major OS in enterprise and desktop computing environments. So far, Windows dominates the desktop OS market with around 73% of the market share worldwide (Statcounter, 2025) and Windows 10 and Windows 11 power most of the enterprise workstations. Specifically in enterprise deployments, Windows is about 68% of the market for both servers and desktops (6sense, 2025), especially in the United States, accounting for 68.19%, the United Kingdom, 8.30% and India, 7.76%.

There are several reasons for selecting Windows OS as the target for this AI-based system monitor:

1. **Enterprise Dominance**: Windows still dominates the enterprise world, with 82% of enterprise devices still using Windows 10, according to ControlUp (2024; cited in Procurri 2025), as of mid-2024.
2. **Resource Management Challenges**: Intelligent monitoring becomes especially useful in Windows environments due to issues such as DLL management, complexity of the registry, dependency of services, and diverse hardware configurations.
3. **Security Concerns**: Security Concerns: With Windows being the most targeted OS for malware and attacks, proactive system monitoring with AI-driven anomaly detection provides significant security benefits.
4. **Hardware Heterogeneity**: Windows is not only running on a wide variety of hardware configurations, but monitoring must also be adaptive and learn the “normal” behavior of different environments.
5. **Performance Optimization Needs**: Resource bloating and performance degradation over time are common challenges in enterprise Windows deployments, where AI powered process management can help.

However, enterprise adoption will continue to be a Windows 10 ecosystem for years to come

because of slow migration cycles and long support periods. This gives a significant and long-term market for system monitoring tools that run on a Windows platform.

2.5 INTEGRATION OF AI APIS (GEMINI) AND SYSTEM METRICS LIBRARIES (SYSINFO)

While there are plenty of previous works focusing on ML model engineering for anomaly detection, there are less papers on how anomaly detection can be integrated with natural-language summarization APIs for system monitors. The Gemini API (from Google) can create a human readable summary of a structured or semi-structured input, so the system monitor can give actionable insight (not just the numbers). In this implementation:

1. The sysinfo crate gathers various Windows OS metrics (cpu usage, memory usage, disk usage, and process counts) using native Rust system calls.
2. The metrics are presented as a prompt and passed on to the Gemini API for summarizing (e.g., "Summarize these system metrics in 3 to 4 concise bullet points…").
3. Gemini returns a list of bullet points (the metadata has been trimmed off for brevity).
4. The summary is displayed or logged in the 4. The summary is displayed or logged in Rust binary and may be automated (e.g., closing processes that consume a lot of resources).

This type of integration (metric-collection + NL summarisation + process management) is fairly new and will fill a void of "human readable insight" in system monitoring. This work takes the idea of full-stack AI-based performance tracing of ML systems from Xu et al. (2025) and applies it to general Windows OS system and process monitoring to provide operationally useful text from a summarization API.

2.5.1 RATIONALE FOR GEMINI API SELECTION

The reasons it makes sense that Google has opted for the Gemini API over other options, such as OpenAI's GPT models, Anthropic's Claude, or local models are justified by:

- **Cost-Effectiveness**: Gemini provides competitive API call prices and has a generous free tier that's ideal for monitoring individual systems.
- **Low Latency**: Gemini's infrastructure ensures low latency essential for near-real-time monitoring applications.

- **Multimodal Capabilities**: his project relies on text-based summaries, but Gemini has built-in support for multimodal data, which will allow future improvements, such as interpreting screenshots or conducting graph analysis.
- **Context Window**: Gemini supports a long enough context window as to not be limited by truncation for comprehensive system metrics.
- **Integration Simplicity**: Gemini's REST API and Rust client library ecosystem enable seamless integration, avoiding cumbersome authentication processes.

- There was also consideration of alternative approaches but these were not followed: **Local LLMs** (e.g., Llama, Mistral): They would need a lot of resources for the system, which is not the point of a lightweight monitoring tool.
- **OpenAI GPT**: The API pricing and rate limiting mechanisms are more intricate with OpenAI GPT, which may mean it is less desirable for frequent monitoring calls.
- **Rule-Based Summarisation**: Not possess the contextual knowledge and natural language skills that are typical of LLMs with Rule Based Summarisation.

2.5.2 SECURITY AND PRIVACY CONSIDERATIONS

Using system metrics on external APIs (Gemini) requires valid concerns of security and privacy that should be addressed:

- **Data Minimisation**: No sensitive information such as file contents, password, personal information is sent through the system – only aggregated data (cpu %, memory usage, process names).
- **Anonymisation**: Process names are added, users, file paths and command line arguments aren't added to API requests.
- **Local-First Option**: The architecture can be designed to support a “local mode,” which provides metrics without making API calls, especially in security-sensitive environments.
- **Encryption**: HTTPS/TLS encryption is used for all API communications, so that the data being transferred is encrypted.
- **Enterprise Deployment Considerations**: For enterprise deployments, the tool can be configured to go through a Proxy or LLM deployed behind Company Firewalls.
- **Compliance**: Industry users (healthcare, finance) in regulated industries should consider whether they must comply with data governance policies by transferring data

even if they anonymise system metrics to the cloud services (e.g., HIPAA, GDPR).

As for security, these points highlight the importance of transparency and the option of privacy settings in actual deployments with AI-driven monitoring solutions. This approach will involve collection of metrics using Rust, collection of safe concurrency using Rust, metric analysis using AI using Gemini, and the recommendation and management using Rust.

2.6 RELATED WORKS

1. Lindemann et al. (2021) surveyed deep learning for time-series anomaly detection, outlining how LSTM-based systems can capture longer-term dependencies in telemetry data. Their work, published in *Computers in Industry*, provides foundational understanding of temporal pattern recognition in system metrics. Lindemann et al. (2021) reviewed deep learning approaches for anomaly detection in time series, explaining the use of LSTM based architectures for detecting long-term anomalies in time series, such as those found in telemetry data. The paper they published in Computers in Industry gives a good foundation of knowledge in temporal pattern recognition in system metrics.
2. Xu, Xie & Chen (2025) suggest a non-instrumented performance-tracing system for ML environments based on Gaussian Mixture Models and eBPF counters, called eACGM. This project's focus is on Windows OS, processes -- that is, a narrower but similar focus than that used in that work, which emphasizes hardware-level metrics and large-scale clusters.
3. Stahmann (2025) investigated how anomaly detection can enhance the control process in industrial systems, demonstrating the improvement with AI based real-time anomaly detection.
4. Survey of AI-based Anomaly Detection in IoT and Sensor Networks was done by De Medeiros et al., 2023. Their paper in the journal Sensors focused on the research published in the last four years (2019-2022), which showed how machine learning (ML) and deep learning (DL) are widely adopted methods for anomaly detection in devices and networks. Although the domain is different to desktop systems, many of the concepts of unbalanced data, and multivariate telemetry, are applicable to system metrics as well.
5. Kohli et al. (2025) did a survey of deep-learning for network anomaly detection—this is a networking topic, but many of the same techniques (multivariate time-series,

forecasting) can be used in system monitoring.

6. Bugden and Alahmar (2022) analysed Rust's safety and performance characteristics, demonstrating through benchmarking research that Rust achieves overall superiority over established languages in systems programming contexts. Their work validates the choice of Rust for performance-critical monitoring tools Kohli et al. (2025) did a survey of deep-learning for network anomaly detection—this is a networking topic, but many of the same techniques (multivariate time-series, forecasting) can be used in system monitoring.

These studies confirm the trend of AI tools to be used for monitoring but do not go so far as end-user summarisation or process management with Rust binaries.

2.7 EXISTING WINDOWS MONITORING TOOLS AND THEIR LIMITATIONS

In order to understand the contribution made by this project it is important to briefly review current options for monitoring Windows based systems and their limitations:

2.7.1 BUILT-IN WINDOWS TOOLS

Task Manager: Windows Task Manager offers a simple way to monitor CPU, memory, disk and network usage, in real time, at the process level. It has, however, the following limitations:

- No historical trend analysis or pattern recognition
- Alerts based on thresholds (no anomaly detection)
- Limited automation capabilities
- Lack of predictive insights and recommendations
- Purely reactive interface: one that needs manual interpretation

Resource Monitor: Provides additional details in its metrics, including per-process file and network activity. Limitations include:

- The interface is complex and hard to use for non-expert users
- No AI generated suggestions or natural language summaries
- Lacks automated remediation capabilities
- No learning across sessions

Performance Monitor (perfmon): Collects and logs vast amounts of data on metrics and customizable data collector sets. However:

- A steep learning curve and experience needed to setup properly

- Raw data output needs manual analysis and interpretation
- No automatic anomaly detection or predictive features
- Ease of use of the legacy interface and complex report generation

2.7.2 THIRD-PARTY MONITORING SOLUTIONS

Sysinternals Suite (Process Explorer, Process Monitor): Industry-standard advanced monitoring tools with deep system insights. Limitations include:

- Provide advanced users with advanced options and troubleshooting information specific to their problems.
- No pattern recognitions or automated insights with A tools
- Needs a lot of user knowledge to analyze the data
- Lack of automatic features and lack of natural language interface

Enterprise monitoring platforms, such as Commercial Solutions (SolarWinds, PRTG, Datadog). These platforms offer a wide range of features. However:

- Individual user / small business is not feasible due to the high cost
- The administration model is complex and calls for dedicated administrators
- There is a tendency to over-engineer for monitoring a single machine
- Cloud dependent system designs with privacy issues
- The absence of lightweight, portable and single-executable deployment.

2.7.3 GAP ANALYSIS

This project has been created to tackle some of the prevalent gaps in current windows monitoring tools:

1. Raw Metrics vs Intelligent Insights: These are missing today, but intelligent insights here, with AI generated natural language summaries of the state of the system and suggested actions as well.
2. No Pattern Learning: Existing solutions apply static thresholds and rules, while this project relies on Gemini's context-awareness to detect any abnormal pattern based on the normal pattern.
3. Poor Accessibility: Traditional monitoring tools are difficult to understand and comprehend without technical expertise, whereas natural language summaries make

system monitoring easily accessible to non-technical users.

4. Limited Automation: Many tools are intended for visualization and alerting, and don't include process management capabilities that this project offers (such as termination of resource-heavy processes).
5. Resource Overhead: Many enterprise solutions consume a lot of system resources, but this Rust-based tool has a minimal resource overhead due to native compilation and efficient design.
6. Deployment Complexity**:** Enterprise tools need to be installed, configured and may need to work on a network, this project is a portable executable with no requirements.

This AI-powered system monitor is a new fusion of low-level native performance (Rust), real-time system data (sysinfo), and natural language intelligence (Gemini API) designed specifically for Windows OS.

2.8 META-ANALYSIS TABLE

Study	**Focus Area**	**Methodology**	**Key Findings**	**Relevance to Topic**
Yu & Wang (2024)	Bug analysis in Rust	Empirical study encompassing a comprehensive analysis of over 10,000 bug reports from Rust projects.	Bugs mostly semantic; unsafe code manageable	Validates Rust's reliability in system tools
Jiang et al. (2023)	Safety in Rust systems	Methodology (Thetis) for encapsulating unsafe code	Reduces unsafe operations by 40–45%	Supports secure system monitor development
Zakopal et al. (2024)	Real-time FPGA monitoring	Hardware/software integration using Rust GUI	Demonstrated low-resource, real-time signal monitoring	Validates Rust in real-time system diagnostics

Sinha et al. (2025)	File search on Windows	Parallelized indexing with Rust	10x speed over Windows File Explorer	Shows performance benefits of Rust on Windows
Minin (2025)	Rust-Python integration	Performance benchmarking using PyO3	Achieves concurrent AI task execution	Facilitates Rust-AI hybrid monitoring systems
Hardin (2023)	Formal verification	ACL2-based verification of Rust code	Fully verified memory-safe data structures	Enhances confidence in AI system monitor correctness
Yosifovich et al. (2017)	Windows architecture	In-depth OS internal documentation	Foundation for accessing Windows APIs	Essential for Rust-Windows API Integration
Assi et al (2022)	Performance diagnostics	WPT-base root cause analysis	Identified performance bottlenecks and improved resource use	Guides the system monitor design and optimization
Gäher et al. (2024)	Code verification in Rust	RefinedRust type system verified in Coq	Ensures safety in both safe and unsafe code	Enables high-assurance Rust monitoring applications

Table 2. 1

CHAPTER 3

3. METHODOLOGY

3.1 RESEARCH DESIGN AND APPROACH

The method used in this study was design science research (DSR) that focuses on the design and testing of new and novel artifacts to address practical problems (Hevner et al., 2004). The research took an iterative development process involving three stages:

1. Requirements Analysis and Design Phase: System monitoring problems Identification and Architecture design.
2. Implementation Phase: Development of the monitoring and AI analysis modules using Rust
3. Evaluation Phase: System performance testing and validation in different operational conditions

Through this iterative process, one could continuously improve the system by repeating the test iterations and refining the system and using feedback from the users to make sure that it meets the technical requirements and the usability needs.

3.2 SYSTEM ARCHITECTURE OVERVIEW

To achieve this, the developed system was structured in three major modules, following a modular design to meet the requirements of scalability, maintainability and separation of concerns:

Module	**Function**	**Status**
Monitoring Module	Gathered system metrics like CPU, memory, disk and network with native rust libraries.	Fully Implemented
AI Analysis Engine	Used a cloud AI model (Gemini API) to understand, identify anomalies, and generate natural language answers from metrics.	Fully Implemented
Process Manager (Conceptual)	To be used by the AI to reprioritize or terminate processes, however, this was only partially effected.	Conceptual Design

Table 3. 1

3.2.1 ARCHITECTURAL DESIGN RATIONALE

The system was designed as a pipeline with unidirectional transfer of data from the collection point through analysis to presentation. The choice of this design was influenced by several factors:

- Modularity: Each component can be developed, tested, and maintained independently
- Extensibility: New monitoring metrics or analysis methods can be added without restructuring the entire system
- Performance: Rust's zero-cost abstractions enable efficient data flow with minimal overhead
- Fault Tolerance: Failures in one module (e.g., API unavailability) do not crash the entire system

Modular architecture allows for seamless data transfer between modules. Real-time metrics were gathered and processed in Rust and sent to the AI analysis engine for interpretation. The result is an analytical output that gives them actionable insights that they use as the foundation for "intelligent" including decision making in the process management framework.

3.3 SYSTEM DEVELOPMENT AND IMPLEMENTATION

3.3.1 MONITORING MODULE

To collect live system performance data from Windows OS, the **monitoring module** has been implemented in Rust. This part is the backbone of the system and is used to collect accurate and current telemetry data.

3.3.1.1 TECHNOLOGY SELECTION: SYSINFO CRATE

Rust's sysinfo crate (version 0.35.2) was chosen as the main system monitoring library for several critical reasons:

1. Native Rust Implementation: The sysinfo crate is built entirely in Rust, giving it direct access to system APIs through zero-cost abstractions. This eliminates the overhead that comes with Python's psutil, which typically relies on inter-process communication or Foreign Function Interfaces (FFI) to achieve similar functionality.

2. Cross-Platform Compatibility: The crate offers the same API on Windows, Linux and macOS platforms, using native system calls (WinAPI on Windows, procfs on MacOS).
3. Performance Characteristics: The crate was selected because it had the least overhead during metric collection, making it suitable for continuous monitoring scenarios without having significant impact on system performance.
4. Memory Safety: As a Rust-native library, sysinfo benefits from Rust's memory safety system, preventing memory leaks and buffer overflows that could compromise a long-running monitoring agent.
5. Rich Metric Coverage: The crate offers extensive access to system resources including CPU usage (per-core and aggregate), memory utilization (RAM and swap), disk I/O statistics, network transfer rates, and detailed process information.

3.3.1.2 METRICS COLLECTED

The monitoring module was designed to collect the following system metrics at configurable intervals (default: 5 seconds):

CPU Metrics:

- Overall CPU usage percentage (aggregate across all cores)
- Per-core CPU utilization
- Number of logical and physical processors
- CPU frequency (when available)

Memory Metrics:

- Total physical RAM
- Used memory percentage
- Available/free memory
- Swap memory usage (if configured)

Disk Metrics:

- Read/write operations per second
- Disk space utilization per partition
- I/O transfer rates (MB/s)

Network Metrics:

- Data received/transmitted (bytes)
- Network interface statistics

Data Collection Implementation

System struct from the sysinfo crate was used to implement the metric collection process in Rust. This is the outline structure in which the implementation took place:

```
use sysinfo::{System, SystemExt};
// Initialize system object
let mut sys = System::new_all();
sys.refresh_all(); // Initial refresh to populate data
// Periodic refresh in main loop loop {
sys.refresh_cpu(); sys.refresh_memory();
// Extract metrics
let cpu_usage = sys.global_cpu_info().cpu_usage();
let memory_usage = (sys.used_memory() as f64 / sys.total_memory() as f64) * 100.0;
// Wait for next collection interval
std::thread::sleep(Duration::from_secs(5));   }
```

Data was gathered and temporarily stored in memory for real time analysis. Selected outputs were deserialized back to JSON and CSV using the serde_json and csv crates for debugging, validation and longitudinal analysis.

3.3.1.4 VALIDATION USING PSUTIL

Python's psutil library has been used as a validation reference during the early development and testing stages, to guarantee the accuracy and consistency of metrics gathered by the Rust implementation. This approach of double measurement was for two reasons:

1. Accuracy Verification: Verify the accuracy of CPU, memory and disk metrics of sysinfo vs psutil and find any discrepancies
2. Baseline Establishment: The goal is to use the well-established implementation of baseline for validation of the Rust based system.

The two implementations were run concurrently and compared against one another at the same times to perform cross validation testing. Any discrepancies were examined to see if they were due to timing differences, API differences or actual measurement errors.

This component was created with high efficiency and safety in mind, and uses Rust's

concurrency model (parallel metric collection with std::thread) and zero cost abstractions to keep the monitoring agent from adding excessive system overhead.

3.3.2 AI ANALYSIS ENGINE

The core of the project was the AI Analysis Engine that analyzed and translated the raw telemetry data into valuable insights and recommendations for human use. Instead of training a custom local machine learning model, which would need significant computational power, training data, and expertise, the system was connected to Google's very powerful large language model (LLM), Gemini, which was tailored for reasoning and natural language generation.

3.3.2.1 RATIONALE FOR GEMINI API INTEGRATION

Several reasons were given for the use of the Gemini API, rather than alternatives:

1. Cost-Effectiveness: Gemini has generous limits for its free tier to accommodate monitoring applications that are used by individual users and Gemini 2.5 Pro has a limit of 15 requests per minute (RPM) and 1,500 requests per day. This equates to 288 queries every day, which is within the free tier limits for a monitoring tool that has a 5-minute query interval.
2. Low Latency: API selected because of the latencies of its responses, which are acceptable to near real time monitoring applications of operational value where delays of 1-2 seconds do not materially affect the value of the monitoring.
3. Contextual Understanding: Gemini's language model can understand metric relationships contextually as opposed to having to define thresholds beforehand (e.g., knowing when a workload is compute-bound vs. I/O-bound by analyzing the relationships between metrics).
4. Natural Language Output: The API produces natural language summaries that can be easily understood by anyone, not just IT experts.
5. No Local Model Training Required: There is no need for labeled training datasets, GPUs or machine learning skills.

3.3.2.2 DATA PROCESSING AND PROMPT ENGINEERING

The workflow for the analysis with AI was structured around the following consecutive steps:

Stage 1: Data Aggregation

The Rust monitoring module extracted the rust metrics, which were then converted to a structurally represented text. In this system, uniformity was maintained by adopting template method:

System Performance Summary:

1. CPU Usage: {cpu_percent}%
2. Memory Usage: {memory_percent}% ({used_memory}GB / {total_memory}GB)
3. Disk Usage: {disk_percent}%
4. Top CPU Consumers: {top_processes_by_cpu}
5. Top Memory Consumers: {top_processes_by_memory}

The concept of this template-based approach was to ensure that the input was consistent and well structured.

Stage 2: Prompt Engineering

The structured data was carefully embedded within crafted prompts designed to guide Gemini toward the intended analysis goals:

1. Discuss and answer the following Windows system measurements and include:
2. Analyze the following Windows system metrics and provide:
    - Brief overall system health assessment (1-2 sentences)
    - Any anomalies or worrying trends identified (bullet points)
    - Specific optimization suggestions (bullet points)
    - Priority level (Low/Medium/High) based on performance impact Format response as:

      HEALTH: [assessment]
      ANOMALIES: [bullet list or "None detected"]
      RECOMMENDATIONS: [bullet list]
      PRIORITY: [Low/Medium/High]

Stage 3: API Interaction

The system was designed to send formatted prompts to Gemini 2.5 Pro using Rust's reqwest crate making async HTTP requests through REST API:

```
use reqwest;
use serde_json::json;
let client = reqwest::Client::new();
let response = client
    .post("https://generativelanguage.googleapis.com/v1beta/models/gemini-1.5-
flash:generateContent")
    .header("Content-Type", "application/json")
    .header("x-goog-api-key", api_key)
    .json(&json!({
        "contents": [{
            "parts": [{"text": formatted_prompt}]
        }]
    }))
    .send()
    .await?;
```

Error handling added to gracefully deal with failures in the API:

- Network timeouts: Retry with exponential backoff (maximum 3 attempts)
- Errors that indicate that the rate limit has been exceeded(429): Queue requests with delay
- API unavailability: If the API is not available, you can choose to show the metrics without AI analysis

Stage 4: Result Processing and Display

The response from Gemini was encoded in JSON with natural language text, and was intended to be able to be picked out to give key elements:

```
let api_response: ApiResponse = response.json().await?;
let analysis_text = api_response.candidates[0].content.parts[0].text;
// Parse structured response
let health = extract_field(&analysis_text, "HEALTH:");
let anomalies = extract_field(&analysis_text, "ANOMALIES:");
let recommendations = extract_field(&analysis_text,
"RECOMMENDATIONS:"); let priority = extract_field(&analysis_text,
```

"PRIORITY:");

This blend of Rust's asynchronous I/O capabilities with Google Gemini's contextual reasoning powers resulted in a hybrid system architecture that is both fast and lightweight while also having the ability to perform sophisticated semantic analysis through cloud-based inference.

3.3.3 PROCESS MANAGEMENT MODULE (CONCEPTUAL DESIGN)

In theory, the Process Manager module was designed to complete the monitoring-analysis-action cycle by automatically or semi-automatically perform remedial actions based on AI generating insights. The architecture and logic of this module was already decided, but It was delayed to be fully implemented due to some technical and security concerns.

3.3.3.1 PROPOSED FUNCTIONALITY

The module was designed for 3 major modes of operation:

1. Advisory Mode (Implemented)
    - Shows AI suggestions for human intervention
    - Offers command suggestions to end or prioritize a process
    - No automatic modifications to the system state
2. Semi-Automated Mode (Partially Implemented)
    - Asks user if they want to make recommendations when asked
    - Enables the use of security measures to prevent the closure of important system processes
    - Records all activities for auditing purpose
3. Fully Automated Mode (Conceptual Only)
    - Would perform optimizations automatically as per AI suggestions
    - If whitelist/blacklist of processes for automated management would be required.

3.3.3.2 TECHNICAL CHALLENGES AND CONSTRAINTS

There are several technical challenges which prevented full implementation throughout the project period:

1. Windows API Complexity: On Windows operating system, managing processes involves complex API hierarchies (OpenProcess, TerminateProcess, SetPriorityClass) and specific access rights and privilege escalation. It would be a security risk to run the system as an administrator.
2. Safety Considerations: Automated process termination may lead to system instability when critical operating system processes or services are accidentally targeted. It was more difficult to set up a establish a well-designed process classification (critical processes vs. user processes vs. application processes) within the project time constraints.
3. Insufficient Testing Environment: Dedicated machines to thoroughly test the process management functionality would be necessary to prevent disruption of development systems, which were not available for this academic project.
4. Rust FFI Overhead: Rust has two options to interface with windows APIs: winapi or windows crates, but the complexity of implementing the Windows APIs required more time than the project scope.

3.3.3.1 CURRENT IMPLEMENTATION STATUS

The system currently generates recommendations on the manual or automatic management of the process in a structured manner:

RECOMMENDED ACTIONS:
[HIGH] Terminate process “chrome.exe” (PID: 4532) using 45% CPU for more than 30 minutes.
[MEDIUM] Lower priority of "backup.exe" (PID: 2847) - non-critical background process
[LOW] Consider closing unused "spotify.exe" (PID: 1923) - memory optimization

These recommendations may be:

- Performed by hand by system administrators.
- The scripted process is designed to be run by batch, monitored by the user.
- To be extended in future work to get complete AI based automation

In the conceptual design and some aspects of implementation, there is a solid base for future development, showing the feasibility of process management with the help of artificial intelligence, but at the same time, the approach to system safety and user control is emphasized for the current release.

3.3 TOOLS AND TECHNOLOGIES

The system has been designed with a carefully selected technology stack, selected based on the degree of performance, safety, integration and ecosystem maturity. A comprehensive summary of the technologies used are given in Table 3.1.

Component	**Tool / Library**	**Version**	**Purpose**	**Selection Rationale**
Programming Language	Rust	1.75+	Primary implementation language for system-level performance, concurrency, and memory safety.	Memory safety without garbage collection; zero-cost abstractions; strong type system; excellent Windows support.
System Monitoring	sysinfo	0.35.2	Collected system metrics (CPU, memory, disk, processes) efficiently on Windows.	Native Rust implementation; cross-platform; low overhead; comprehensive metric coverage.
HTTP Client	reqwest	0.11.x	Enabled asynchronous API calls to Gemini API.	Async/await support; robust error handling; widely adopted in Rust ecosystem.
JSON Serialization	serde, serde_json	1.0.x	Managed structured JSON request/response data for API communication.	Zero-copy deserialization; compile-time type checking; ecosystem standard.

CSV Export	csv	1.2.x	Logged system metrics to CSV format for external analysis.	Efficient streaming writes; configurable field separators; handles large datasets.
AI Integration	Google Gemini API (1.5 Flash)	v1beta	Performed AI-based interpretation, summarization, and recommendations.	Cost-effective (generous free tier); low latency; strong natural language generation; no local model training needed
Python Testing	psutil	5.9.x	Verified metric consistency during development and testing phases.	Industry-standard system monitoring library; well-documented; served as ground truth for validation.
Target Operating System	Windows 10 / 11	21H2+	Platform for monitoring, testing, and deployment.	Dominant enterprise OS (73% market share); extensive API documentation; target user base.
Development Environment	Visual Studio Code + rust-analyzer	Latest	IDE with Rust language server for development.	Excellent Rust tooling; integrated debugger; Git integration.
Build System	Cargo	1.75+	Rust's package manager and build system.	Dependency management; incremental compilation; cross-compilation support.
Version Control	Git	2.40+	Source code management and versioning.	Industry standard; enables rollback and collaboration.

Table 3. 2

This combination has been chosen to provide an optimal balance between speed (native Rust)

and overheads (runtime overhead). These include compilation (compiled by the system), accuracy (checked against psutil), and AI-powered intelligence (Gemini API), while maintaining minimal resource consumption.

3.4.1 DEVELOPMENT ENVIRONMENT CONFIGURATION

The development environment was set up with the following:

Hardware Specifications:

- Processor: Intel(R) Core i5 (8 cores, 1.60GHz)
- RAM: 8GB DDR4
- Storage: 238GB NVMe SSD
- Operating System: Windows 11 Pro (Build 26100.6584)

Software Configuration:

- Rust toolchain: 1.75.0 (stable-x86_64-pc-windows-msvc)
- Cargo: 1.75.0
- Visual Studio Code: 1.85.x with rust-analyzer extension
- Git: 2.42.x

API Configuration:

- Gemini API Key: This was obtained from Google AI Studio (ai.google.dev)
- API Endpoint: https://generativelanguage.googleapis.com/v1beta/models/gemini-1.5-flash:generateConten
- Rate Limiting: Limited to free tier limits (15 RPM, 1,500 RPD)

3.4.2 DEPENDENCY MANAGEMENT

All Rust dependencies were handled using Cargo.toml:

```
[dependencies]
sysinfo = "0.29"
humansize = "2.1"
chrono = "0.4"
csv = "1.1"
google-ai-rs = "0.1.2"
reqwest = { version = "0.12.23", features = ["json"] }
```

```
serde = { version = "1.0", features = ["derive"] }
serde_json = "1.0.143"
tokio = { version = "1", features = ["full"] }
dotenvy = "0.15.7"
```

This declaration of dependency was needed to guarantee reproducible builds among various development and deployment setting.

3.5 TESTING AND EVALUATION METHODOLOGY

This section explains the systematic method used to test the developed system for its performance, accuracy and usefulness. Evaluation method was designed to evaluate both technical and user experience factors.

3.5.1 TESTING ENVIRONMENT SETUP

To test it, two different Windows machines were planned to be used, to ensure reproducibility and to evaluate performance differences between different hardware configurations:

Test Machine 1 (Development System):

- CPU: Intel(R) Core i5 (8 cores, 1.60GHz base / 1.80GHz boost)
- RAM: 8GB DDR4-2400MHz
- Disk: 238GB NVMe SSD
- Operating System: Windows 11 Pro (Build 26100.6584)
- Network: Gigabit Ethernet

Test Machine 2 (Representative User System):

- CPU: AMD Ryzen 5 3600 (6 cores, 3.6GHz base / 4.2GHz boost)
- RAM: 16GB DDR4-2666MHz
- Disk: 512GB SATA SSD (Crucial MX500)
- OS: Windows 10 Pro (Build 19044)
- Network: Wi-Fi 802.11ac

This dual-machine approach was developed to guarantee the system would be reliable across different hardware combinations that would be standard in enterprise and consumer environments.

3.5.2 TESTING SCENARIOS

The system was to be tested in four different operating conditions all intended to mimic a real-life application.

3.5.2.1 BASELINE TESTING (NORMAL USAGE)

Objective: To set up performance baselines and to confirm that the proper metrics are being captured with standard desktop applications.

Procedure:

1. Run the monitoring tool continuously for 24 hours
2. Carry out daily activities including:
    - Web browsing (Microsoft Edge, Google Chrome)
    - Printing, editing and copying (Microsoft Word, Excel)
    - Video conferencing (Microsoft Teams, Zoom)
    - Email client use (Outlook)
    - Background tasks (Windows Defender, OneDrive sync)

   This will involve background tasks such as windows defender and OneDrive sync
3. Record system metrics every 5 seconds
4. Compare collected metrics against readings from Windows Task Manager for validation

Data Collection:

1. Program integration (how well the program integrates with the monitor)
2. The accuracy of metric collection (deviation from Task Manager values)
3. Installing AI analysis coherence and relevance.
4. The distribution of API response latency.Distribution of latency for API responses.

3.5.2.2 HIGH WORKLOAD TESTING (STRESS CONDITIONS)

Objective: Isolate the system and test how fast and accurate AI responses are in systems under high loads.

Procedure:

1. Execute CPU-intensive workloads:
    - HandBrake (and 4K video files) were used for video rendering.
    - Compilation of software (Large Rust project with parallel compilation)
    - Multi-threaded compression (7-Zip benchmark mode)
2. Execute memory-intensive workloads:

- Run 50+ programs simultaneously 50+ programs at the same time
- The size of the workbook exceeds the limits. The worksheet is too big (>100MB when complex formulas are used.
- It includes various features, such as the ability to host several virtual machines at once. It can run multiple virtual machines simultaneously.

3. Execute disk-intensive workloads:
    - There are no limitations on file size transfers of 10GB+ files.
    - Database indexing operations
    - Sequential write operations - operations that are performed one by one.
4. Do each of the stress tests for 2 hrs and record for 5 seconds.
5. Record the time delay after start of an AI
6. Assess appropriateness of AI recommendations

Data Collection:

- Detection delay (time from detection to identification of the anomaly)
- The relevance of the recommendations (qualitative assessment)
- The proportion of individuals with normal test results who test positive in a stress situation is the false positive rate.
- Good performance during test period.

3.5.2.3 ANOMALY SIMULATION TESTING

Objective: Test AI's capability to identify and analyze artificially added performance degradations.

Procedure:

1. After normal operation, set up 10-minute baseline.
2. Add one particular anomaly at a time:
    - Memory Leak: A Python script that will cause a memory leak due to unbounded list growth.
    - CPU Spike: Infinite loop in background process.
    - Disk Saturation: This is the case when thousands of small files are created and deleted quickly.
    - Process Proliferation: Prolonged processes that spawn 100+ child processes per batch script.

4. Monitor each anomaly for 15 minutes and hold the anomaly for 15 minutes.
5. Capture messages and suggestions from AI detection
6. Rank AI's answers to answer whether they were true/false/partially true.

Data Collection:

- Time to Detection: seconds elapsed from introducing an anomaly to the time the anomaly was detected by AI
- Anomaly type classification accuracy (correct identification of anomaly type).
- Recommendation actionability (Is the user able to solve the problem with the AI recommendations)
- A number of false negative events (i.e., events that were not detected)

3.5.2.4 Longitudinal Reliability Testing

Objective: Evaluate the stability of a system and typical performance over long run duraions.

Procedure:

1. Run monitoring tool for 7 days (168 hours) continuously.
2. Configure automatic restart on crash with logging (yes/no).
3. Run both test machines at the same time.
4. Do spot checks to compare metrics with Task Manager every 24 hours
5. Measure the memory usage of the monitor application over time.
6. Record all successful and unsuccessful API calls.

Data Collection:

- This has been updated to include uptime percentage and crash logs.
- Memory usage trends (look for possible leaks)
- The percentage of successful calls and the fluctuation of latency over time. Success rate and changes in latency over time for API calls.
- The consistency in the accuracy of the meter over the period of testing.

3.5.3 EVALUATION METRICS

The performance of the system was planned to be measured quantitatively as well as qualitatively based on existing software testing and performance evaluation techniques.

3.5.2.1 QUANTITATIVE METRICS

1. **Detection Accuracy**: The proportion of actual system anomalies correctly identified by the AI analysis engine.

$$Detection\ Accuracy = \frac{True\ Positives}{True\ Positives + False\ Positives} \times 100\%$$

Where:

- True Positive = Actual anomaly correctly identified
- False Negative = Actual anomaly missed by AI

2. **False Positive Rate**: The proportion of normal system states incorrectly flagged as anomalous.

$$False\ Positive\ Rate = \frac{False\ Positives}{False\ Positives + True\ Positives} \times 100\%$$

Where:

- False Positive = Normal state incorrectly flagged as anomalous
- True Negative = Normal state correctly identified as normal

3. **Response Latency:** Total time elapsed between metric collection initiation and presentation of AI-generated analysis to the user.

Measurement Method: Timestamp analysis with components:

- Data collection time (sysinfo refresh cycle)
- Prompt formatting time
- API request/response roundtrip time
- Response parsing and display time

4. **System Overhead**: This is the additional CPU and memory resources consumed by the monitoring tool itself relative to total system capacity.

Measurement Method:

- CPU Overhead: Monitor's own CPU usage measured via Task Manager
- Memory Overhead: Monitor's own RAM consumption (Working Set)
- Network Overhead: Total bytes transmitted to/from Gemini API

5. **API Reliability:** This is the ratio of successful API requests to total requests calls.

Measurement Method:

$$API\ reliability = \frac{Successful\ Requests}{Total\ Requests\ Attempted} \times 100\%$$

3.5.3.1 QUALITATIVE METRICS

1. **Insight Quality**: This is the subjective assessment of the clarity, relevance, and usability of AI-generated summaries.

Measurement Method:

a. Three independent evaluators (2 technical, 1 non-technical) rate 50 randomly selected AI outputs

b. 5-point Likert scale assessment across three dimensions:
   - Clarity: Is the analysis understandable?
   - Relevance: Does it address actual system state?
   - Actionability: Can users act on suggestions?

c. Calculate inter-rater reliability using Cronbach's alpha

2. **User Experience Feedback**

Measurement Method:

a. Invite 5 players (one IT user and 3 non-technical users)

b. Every participant is allowed to use the system for 3 consecutive days.

c. Conduct semi-structured interviews assessing:
   - Easy to use and user-friendly interface
   - Increase confidence in AI-generated recommendations.
   - Grow trust in AI recommendations.
   - They believe that the value of the tools is better than what they already have (Task Manager and Resource Monitor).
   - Suggestions for improvement

Comparison with Existing Tools

Measurement Method:

a. Comparing side by side with Windows Task Manager & Resource Monitor

b. Assess differences in:
   - Displaying information (raw data vs. natural language)
   - Actionability of insights

- Ease of interpretation by the user (readability of the user's content)
- The individual or group needs to consider the time required for diagnosis and comprehend system problems.

3.5.4 DATA COLLECTION AND ANALYSIS PROCEDURES

Data Logging:

- All system metrics were automatically recorded in CSV files, with time stamping.
- API responses were downloaded and stored as JSON files for further analysis afterward.
- Application logs logged errors, warnings, and state changes.
- Screen captures taken for documentation at moments of key events

Statistical Analysis Plan:

- All quantitative metrics: descriptive statistics (mean, median, standard deviation)
- Use histograms and plots of time-series to visualize the distributions of the metrics.
- Inter-rater reliability calculations for qualitative assessments were conducted.
- Performance comparisons between testing scenarios and appropriate statistical testing (t-tests, ANOVA as appropriate).

Quality Assurance:

All testing procedures were recorded in detailed test protocols.

Before the execution of the test cases, they were reviewed by the project supervisor.

Data collection instrument, (Logging code) was tested with pilot which was done to validate the data collection instrument.

The raw data were retained for the purpose of reproducing and verifying results.

3.6 ETHICAL CONSIDERATIONS

3.6.1 PRIVACY AND DATA PROTECTION

The system is designed keeping in mind privacy:

1. **Data Minimization:** No sensitive user data, file contents or personal information were passed through to the Gemini API, only aggregated system metrics (CPU percentage, memory percentage process names) were passed through.

2. **Local Processing:** Raw metric collection and storage was done on the local machine only

3. **Informed Consent:** User Testing Participants were informed that the anonymized system metrics will be transmitted to Google's Gemini API and gave written consent.

4. **Data Retention:** Data is collected securely and will be discarded at the end of the project and after evaluation.

3.6.2 SECURITY CONSIDERATIONS

1. API Key Protection: The Gemini API key was used in environment variables, not hardcoded in the source code.

2. Administrator Privileges: The API failed or the network connection failed, but the system degraded gracefully instead of crashing or revealing any vulnerabilities.

3. Fail-Safe Design: The API failed or the network connection failed, but the system degraded gracefully instead of crashing or revealing any vulnerabilities.

3.7 LIMITATIONS OF THE METHODOLOGY

There were several limitations to the methodology:

1. Limited Testing Scale: Testing was performed on a limited number of machines using similar Windows configurations, and may not reflect the broad range of real world deployment scenarios.

2. Controlled Testing Environment: Tests used anomalies simulation testing which may only be artificial or controlled scenarios that do not cover all complex performance anomalies.

3. Small User Testing Sample: Results from user experience testing with 5 users may not be indicative of users at large.

4. API Dependency: System's smart capabilities depend on the availability of external APIs, thus adding dependency on third-party service reliability.

5. Short-Term Longitudinal Testing: Testing which takes place for seven days or more may not show problems that occur over months of deployment.

6. Windows-Only Evaluation: Testing was done on the Windows OS only, which reduces the generalizability of the findings to other platforms as sysinfo is cross-platform.

The limitations are discussed in Chapter 5 (Conclusions and Recommendations) and suggestions are given for further work.

CHAPTER 4

4 RESULTS AND DISCUSSION

4.1 OVERVIEW

The results of implementation and testing the AI-Driven System Monitor for Windows OS are presented in this chapter. The project was a success in creating a working system that leverages the system programming abilities of Rust along with Google's Gemini API for intelligent interpretation of system metrics. Simple usage scenarios were used to ensure that the system had the ability to capture accurate metrics, produce meaningful AI-driven information and be able to present the information in human readable format. The outcomes are structured into four areas: (1) system implementation verification (2) metric accuracy validation (3) analysis of the outputs of the AI system and (4) discussion of findings and limitations.

4.2 SYSTEM IMPLEMENTATION VERIFICATION

4.2.1 SUCCESSFUL DEVELOPMENT OF CORE COMPONENTS

Based on the development, a working system monitor is created, comprising two working modules

1. **Monitoring Module**: Implemented successfully with Rust's sysinfo crate (v0.35.2), which is able to gather various system metrics such as processes, disk usage, memory usage, and CPU usage. There was no crash while running the module in any of the testing sessions.
2. **AI Analysis Engine**: Seamlessly integrated with Google's Gemini 1.5 Flash API, now it can accept structured metric data and generate natural language summaries. The integration included the use of the Rust crates reqwest and serde_json for managing HTTP requests and JSON responses respectively.

4.2.2 SYSTEM OPERATION WORKFLOW

The system that has been implemented works as follows:

1. To get the current system metrics, the monitoring module uses the sysinfo to gather the metrics.
2. The metrics are presented as a text prompt in a structured way.
3. Gemini API responds with the prompt's completion.
4. The completed prompt is returned as a response to the HTTP request to the client. Gemini evaluates the numbers and provides a natural language summary

5. Summary is parsed and presented to the user with colour coded priority level.

The process was carried out without any issues during testing, proving that it is possible to integrate cloud-based AI analysis with native Rust monitoring.

4.2.2 DEPLOYMENT CHARACTERISTICS

The resulting Rust binary had the following properties:

1. No Additional Dependencies: No additional files or installation were needed to run it
2. Installation: It was easily installed on both Windows 10 and Windows 11 operating systems
3. Quick Startup: Application initialized and started monitoring in seconds Portable: The executables were easily able to be "copied" to other Windows machines and run immediately

4.3 METRIC ACCURACY VALIDATION

To ensure that the system gathers accurate data, measurements were taken from the Rust implementation and compared with the **Windows Task Manager** as a reference standard.

4.3.1 VALIDATION METHODOLOGY

The validation process involved:

1. Running the Rust monitoring tool and Windows Task Manager simultaneously
2. Analyzing metrics over same points in time
3. Comparing the values of CPU percentage, memory usage, and disk usage
4. Taking note of any significant discrepancies

4.3.2 VALIDATION RESULTS

1. **CPU Usage**: The CPU percentages gotten from the Rust tool were fairly correct compared to the Task Manager readings. Minor differences were noted (usually 1-3%), and can be explained by:
    - The variability of the sampling interval (time) between tools.
    - Task Manager updating according to its own refresh rate
    - The random fluctuations in CPU load between measurements

Example Comparison:

Time Point	Task Manager CPU	Rust Tool CPU	Difference
Test 1	23%	24%	1%
Test 2 (High Load)	87%	89%	2%
Test 3	15%	14%	1%

Table 4. 1

2. **Memory Usage**: The memory readings were in good agreement with Task Manager. Total memory was matched and used memory within reasonable limits.

Example Comparison:

Time Point	Task Manager Memory	Rust Tool Memory	Match Quality
Test 1	3.4GB / 8GB (42%)	3.4GB / 8GB (42%)	Very Close
Test 2	6.1GB / 8GB (76%)	6.2GB / 8GB (77%)	Very Close
Test 3	4.1GB / 8GB (51%)	4.0GB / 8GB (50%)	Very Close

Table 4. 2

3. **Disk Usage**: Like the memory usage percentages, disk space percentages compared to each other were in line with the metrics displayed in task manager, indicating that the metrics were being collected accurately

4.3.2 VALIDATION CONCLUSION

The assessment showed that the sysinfo crate will give precise system metrics which can be used for monitoring. The minor variances found (1-3%) are within normal limits for system monitoring tool and have no influence on the viability of the system.

4.4 AI ANALYSIS OUTPUT EVALUATION

4.4.1 PROMPT DESIGN

The AI responses were formalized by employing a structured prompt template.

- `Anomalies or Concerning Patterns: Enumerate major abnormalities or warning signs (bullet points).`
- `Optimization Recommendations: Identify specific,`

```
   actionable recommendations (bullet points).
 • Priority Level: Assign priority to the overall performance
   (Low/Medium/High).
 • Be brief per section (100 words maximum). System Metrics:
       - CPU Usage: {cpu_percent}%
       - Memory Usage: {memory_percent}% ({used_memory}GB /
         {total_memory}GB)
       - Disk Usage: {disk_percent}%
       - Network Activity: {bits transmitted}+{bits received}
       - Top CPU Consumers: {top_processes_cpu}
       - Top Memory Consumers: {top_processes_memory}
```

This is a format that was created to help Gemini create structured and actionable summaries instead of a generic response.

4.4.1 SAMPLE OUTPUTS FROM TESTING

The following examples show the output of the system in various situations:**Example 1:** Normal System Operation (Low Priority)

System Metrics Collected:

- CPU Usage: 18%
- Memory Usage: 38% (6.1GB / 16GB)
- Disk Usage: 42%
- Network Activity: 398.00 B
- Top CPU: Code.exe (47.57% CPU)
- Top Memory: chrome.exe (6738 MB)

Gemini AI Output:

```
Anomalies or Concerning Patterns:
 • There are no significant anomalies. System operating in
   normal range.
 • Optimization Recommendations:
     o System is running smoothly. If there is a lack of
       memory in the future, you should consider closing
       unused browser tabs.
```

    - The current levels of resource use are satisfactory.

Priority Level: Low

Observation: The AI has accurately determined a "Low" priority for a healthy system state. All the recommendations were noncritical, consistent with the condition of the system.

Example 2: High CPU Load Scenario (High Priority)

System Metrics Collected:

- CPU Usage: 68.81%
- RAM Usage: 81.64% (6.41 GB)
- Disk Usage: 70.40% (333.58 GB)
- Top CPU: Code.exe (478.57% CPU)
- Top Memory: chrome.exe (608952 MB)

Gemini AI Output:

Anomalies or Concerning Patterns:

- High RAM Usage (81.6%): May cause system instability and performance issues.
- Optimization Recommendations:
    - Identify and kill unnecessary memory consuming processes. Research issues or problems with memory leaks or inefficient configurations.
    - If the workload is anticipated, consider increasing the allocated RAM.

Priority Level: High

Observation: The AI was able to correctly determine the chrome.exe process as the primary resource consumer and contextual suggestions. Did not say terminate, but rather: "Monitor" or "Pause" if not urgent (to avoid losing work).

4.4.2 OUTPUT CONSISTENCY

During the testing: The AI has always:

- Used a structured approach (Anomalies, Recommendations, Priority)

- Provided answers in easy-to-understand words Set priorities in accordance with the severity of the system conditions.
- Generated outputs within 1-2 seconds of API requests

In some instances (around 1 in 20 requests), the output format of the AI was slightly different, but the structured prompt was able to steer the majority of responses towards the desired format.

4.4.3 USER FEEDBACK ON AI OUTPUTS

Sample AI outputs were reviewed by multiple testers (technical and non-technical users).
Key feedback included:

Positive Feedback:

- "The explanations provided are much clearer than the Task Manager numbers."
- "It tells me what the actual issue is and not just the percentages, so I like that."
- "Help knowing if to worry or not (Priority levels)"
- "The suggestions are specific and active"

Critical Feedback:

- "Sometimes the suggestions are very general (e.g., 'close unused applications')"
- "More explanation about WHY something is concerning, would be helpful"
- "Wish it could close the processes, rather than just recommending it."

Overall, testers were impressed with the natural language input interface, and especially favored by non-technical users who felt that the system was easier to use than conventional monitoring applications.

4.5 TESTING SCENARIOS AND OBSERVATIONS

4.5.1 NORMAL USAGE TESTING

Scenario: The monitor running while working normally on computer (browsing the web, editing documents, reading e-mail).

Observations:

- The system was able to properly detect normal operation with "Low" priority ratings.
- The percentage of CPU and memory were the same as in Task Manager.
- No noticeable decrease in computer speed was observed with the use of the tool.
- In normal operation, no AI recommendations were provided and they were optional.

Conclusion: This system works well in the normal operation and doesn't disturb the normal working flow.

4.5.2 HIGH WORKLOAD TESTING

Scenario: Operating resource-intensive processes such as video rendering, large file compression and running multiple applications while using the computer.

Observations:

- The AI was able to correctly identify increased resource consumption.
- Priority levels were appropriately escalated as "Medium" or "High".
- Recommendations accurately identified the processes that use the resources
- The monitoring tool remained in operation even during system stress.

Conclusion: The system is able to correctly detect and describe high load situations.

4.5.3 ANOMALY SIMULATION TESTING

Scenario: Deliberate conditions that cause problems (memory leaks, infinite loops, too many processes).

Observations:

- In most cases, the AI identified abnormal patterns.
- Among other features, memory leak detection was found to be very effective (high memory, low CPU)
- The infinite loop warnings were displayed as soon as the CPU started spiking due to infinite loops.
- Some subtle anomalies (such as disk I/O) were more difficult to describe by the AI.

Conclusion: The system is capable of detecting common anomalies in performance, but is still not very good at detecting complex, multi-factors.

4.6 DISCUSSION OF FINDINGS

4.6.1 ACHIEVEMENT OF PROJECT OBJECTIVES

The project has successfully met its main objectives including:

1. Real-time Monitoring: The system was able to get accurate system metrics through Rust's sysinfo crate, values are very similar to Task Manager.
2. AI-Driven Analysis: The AI-driven analysis, when integrated with the Gemini API, was effective in converting raw metrics into natural language summaries and providing contextual recommendations.
3. Human-Readable Output: User feedback indicated that the natural language output was more accessible than traditional numerical output.

4. Cross-Platform Compatibility: The Rust implementation resulted in a portable.
5. An executable for Windows 11 and Windows 10 that does not require installation.

4.6.2 STRENGTHS OF THE SYSTEM

Technical Strengths:

- Accuracy: Metric collection was reliable, and comparable to existing tools
- Efficiency: There was no significant system overhead in the Rust implementation.
- Scalability: Ability to seamlessly scale with the expansion of the cloud AI environment.
- Interoperability: Efficient interaction between cloud AI analytics and native system monitoring.
- Portability: Single executable deployment makes distribution and testing easier.

Functional Strengths:

- Contextual Understanding: The AI showed not just the numbers, but the interpretation of the metric patterns.
- Appropriate Prioritization: Priority level assignments for most of the systems were appropriate for their degree of severity.
- Actionable recommendations: Suggestions were specific and implementable by users
- Accessibility: The natural language interface was much easier to understand for non-technical users.

4.6.3 LIMITATIONS AND CHALLENGES

Technical Limitations:

1. Network Dependency: AI analysis depends on internet connectivity. If the API isn't available, then only raw metrics can be shown (this is a fall-back feature).
2. API Latency: API round trip analysis takes 2-5 seconds and is not suitable for millisecond analysis.
3. No Process Management: No suggestions for the action are made and the action itself cannot be executed (e.g., no suggestion of terminating processes, and no ability to actually terminate processes). This is a design constraint for complexity and safety concerns.

Functional Limitations:

1. In about 20-25% of the cases, the AI gave generic recommendations (such as 'close unused applications') instead of specific recommendations.
2. Complex Anomaly Detection: Multi-factor performance problems (such as disk I/O bottlenecks and memory pressure) were incorrectly described or partially described.
3. No Historical Context: The system only checks for snapshots, and does not perform historical trend analysis which can cause it to ignore slow performance degradation.
4. No Baseline Learning: This system is not based on an anomaly detection system as those learn 'normal' behavior over time, but is purely based on the AI's general knowledge without any training specific to the system..

4.6.4 COMPARISON WITH TRADITIONAL MONITORING TOOLS

Advantages Over Traditional Tools:

- Interpretation vs. Presentation: Task Manager provides raw data whereas this system interprets the raw data and its implications
- Accessibility: The non-technical users found the approach of explanation more understandable.
- Contextual Recommendations: Gives users actionable recommendations, instead of asking users to decide on the proper action.

Disadvantages Compared to Traditional Tools:

- No Process Management: Cannot directly terminate, pause, or reprioritize processes, such as using Task Manager to temporarily pause or reprioritize processes
- Network Requirement: The machine needs to be connected to the internet to enable AI capabilities.
- API Dependency: Limited availability of third-party services makes it difficult to use API.
- Latency: The delay required for the changes to appear in the visual view of Task Manager is slower than what occurs in that utility.

Complementary Value Proposition: This is a complementary value system, rather than a substitute for traditional tools, especially advantageous to quick health checks, non-expert users and situations where interpretive context is key.

4.5.6 PRACTICAL IMPLICATIONS

The successful implementation illustrates various practical lessons to be learned:

1. Viability of Hybrid Architecture: The feasibility of Hybrid Architecture (native

monitoring (Rust) and cloud AI (Gemini)) has been demonstrated when creating intelligent system tools without the need for ML expertise or GPU infrastructure.

2. Prompt Engineering Effectiveness: With a well-structured prompt, a general-purpose LLM can generate a domain-specific response without fine-tuning the model.
3. Natural Language as Interface: Natural language monitoring interfaces were well-received by users, especially by non-technical end-users who are not used to using system monitoring.
4. System Tool Development on Windows: The Windows system tools development was done successfully using Rust which yields efficient, safe and portable executables.

4.5.7 AREAS FOR FUTURE ENHANCEMENT

A number of enhancements were identified based on the testing and user feedback:

1. Process Management Integration: To complete the monitoring-analysis-action cycle, actual process control (with the appropriate safety measures) should be implemented.

2. Historical Trending: Adding time-series data collection and visualization would allow the ability to detect trends and gain predictive insights.

3. Baseline Learning: False positives could be lessened by using algorithms that learn the "normal" behavior of the system.

4. Local LLM Option: Offering a local model option would remove the network dependency and privacy issues in sensitive environments. Further refinement of prompts could help to make recommendations more specific and to limit generic output.

5. Multi-Machine Monitoring: This would be useful in enterprise development when it gets extended to monitor multiple machines in a network from a central system.

CHAPTER 5

5.1 CONCLUSION

With increasingly complex and demanding computing systems, there has been an increasing need for intelligent, efficient and reliable system monitoring solutions. This project sought to address this gap, by creating a System Monitor and Process Manager for Windows OS in the Rust programming language with Artificial Intelligence capabilities. It was constructed to combine the security, speed and concurrency benefits that Rust brings to system development with the analytical reasoning skills to offer an innovative, proactive approach to managing system performance. Rust's use was driven by its benefits of memory safety, high performance, and minimal resource consumption requirements, which are crucial for efficient real-time system monitoring. While traditional interpreted languages rely on runtime checks, Rust prevents memory leak and unsafety of accessing threads at runtime. These advantages enabled the development of a lightweight, extremely responsive monitoring engine that could manage extremely low-level system metrics like CPU use, memory consumption, disk input/output, and network activity with extreme precision. To take the monitoring beyond the traditional monitoring, the system was connected with AI-based analytics utilizing the Google Gemini API. This allowed for live system metrics to be processed and structured and human-readable analyses were returned. The AI-generated summaries yielded valuable insights under clearly defined categories – Anomalies or Concerning Patterns, Optimization Recommendations and Priority Level. The project leveraged the cloud-based intelligence of Gemini to safely move complex inference to the cloud, eliminating heavy inference demands from the host system. The process management was not fully implemented in this version, but the system monitoring and intelligent analysis was successfully implemented and tested. The metrics of the system were obtained by the sysinfo crate in rust, and during early development stages, psutil was used for validation. The AI feature always generated short, effective summaries, giving users a real-time point of view on performance problems or inefficiencies. The test results demonstrated the effectiveness of the design, with the operation remaining stable, reporting the metrics accurately, and recommendations made by artificial intelligence clearly. The code illustrates the possibility and benefits of integrating Rust's systems-level optimizations with AI-driven reasoning for real-world performance management. The project is built in a modular fashion, with a Monitoring and an AI Analysis component as well as a (future) Process Management component, giving a good basis for extension. Process control features which are directly encoded, predictive modelling, or even local AI inference, in the form of lightweight Rust-compatible machine learning frameworks, can be included in future releases. Finally, the

overall outcome of this project is a positive step towards the monitoring of intelligent systems in Windows OS. It combines the power of systems programming with intelligent adaptive AI to transform systems management from a reactive to a proactive, self-optimizing approach. These integrations will provide more intelligent, faster and independent computing environments that will be able to remain in peak conditions with a minimal amount of man-up attention as the field changes.

5.2 RECOMMENDATIONS AND FUTURE WORK

The conception of the AI-Driven System Monitor and Process Manager for Windows OS has provided a solid foundation for intelligent system optimization and performance analysis. However, there are a some modifications and future research plans recommended to be made to fully achieve the potential of this innovation.

1. Full Process Management Integration: Future versions are intended to not only monitor but also use Full Process Management Integration to actively manage processes. This will allow the system to prioritize, pause and close resource-heavy applications on its own using AI analysis. This enables the system to progress from being passive to active, to optimize and stabilize when under heavy load.
2. Predictive Performance Analysis: Introduction of machine learning models that can predict these CPU, memory and disk forecasts would allow the system to anticipate and identify potential performance issues. This forecasting would make it a proactive performance watchdog, improving reliability in the cases where uptime and performance are critical responsiveness is paramount.
3. Local AI Inference Capabilities: The implementation relies on the Google's Gemini API model for intelligent analysis, but future work will explore using on-device AI inference with light-weight Rust-ready frameworks like tch-rs (PyTorch bindings for Rust) or burn. This would help to reduce the dependence on external APIs, enhance data privacy, and maintain high level of analysis without compromising the offline experience.
4. Improved Visualization and User interface: Making a graphic table, for example, using Tauri or Yew (Rust) would improve user readability. Real-time charts of CPU, RAM, disk usage and network metrics, along with AI summaries and alerts, could be shown. using a visual interface providing users with better understanding of their systems' health.
5. Cross-Platform Compatibility: This version is designed for the Windows OS, but the modular design would support for Linux and macOS platforms in the future. The

system could be expanded to be a universal cross-platform performance manager, so it could fit into a broader array of computing environments since all the platform-specific detailed were removed.

6. Integration with Cloud Monitoring Systems: New support for cloud monitoring systems synchronization and remote metric reporting could enhance the application's value for the network administrator and enterprise. This would enable a central control of various numbers of Systems and cloud-based services in real-time.

7. Security and Privacy Enhancements: With AI parts managing with system data, it is vital to have robust data security and API protection measures. Future development could include encrypted communications, anonymized telemetry and secure API key the management to protect the user's data.

REFRENCES

1. Balasubramanian, A., Baranowski, M. S., Burtsev, A., Panda, A., Rakamarić, Z., & Ryzhyk,
2. L. (2017). System programming in Rust: Beyond safety. *ACM SIGOPS Operating Systems Review, 51*(1), 94-99. https://doi.org/10.1145/3139645.3139660
3. Bugden, W., & Alahmar, A. (2022). Rust: The programming language for safety and performance. *arXiv preprint arXiv:2206.05503*.
4. Chandola, V., Banerjee, A., & Kumar, V. (2009). Anomaly detection: A survey. *ACM Computing Surveys, 41*(3), 1–58. https://doi.org/10.1145/1541880.1541882
5. Chatterjee, P., & Das, A. (2024). AI-powered anomaly detection for real-time performance monitoring in cloud systems. *International Journal of Scientific Research in Science & Technology, 11*(6), 592-601.
6. De Medeiros, K., Hendrick, P., Monroy, S. A., Jino, M., & Alvarez, M. (2023). A survey of AI-based anomaly detection in IoT and sensor networks. *Sensors, 23*(3), 1352. https://doi.org/10.3390/s23031352
7. Jung, R., Jourdan, J. H., Krebbers, R., & Dreyer, D. (2017). RustBelt: Securing the foundations of the Rust programming language. *Proceedings of the ACM on Programming Languages, 2*(POPL), 1-34. https://doi.org/10.1145/3158154
8. Kim, G.-Y., et al. (2022). A study on performance metrics for anomaly detection. *Electronics, 11*(8), 1213. https://doi.org/10.3390/electronics11081213
9. Kohli, M., et al. (2025). A comprehensive survey on techniques, challenges and trends in deep learning for anomaly detection. *SN Applied Sciences.* Advance online publication.
10. Lindemann, B., Maschler, B., Sahlab, N., & Weyrich, M. (2021). A survey on anomaly detection for technical systems using LSTM networks. *Computers in Industry, 131*, 103498. https://doi.org/10.1016/j.compind.2021.103498
11. Liu, Y., et al. (2024). [Add full citation if available from original document]
12. Matsakis, N. D., & Klock, F. S. (2014). The Rust language. *ACM SIGAda Ada Letters, 34*(3), 103-104. https://doi.org/10.1145/2692956.2663188

13. Moharam, M. H., Hany, O., Hany, A., Mahmoud, A., & Mohamed, M. (2025). Anomaly detection using machine learning and adopted digital twin concepts in radio environments. *Scientific Reports, 15*, 18352. https://doi.org/10.1038/s41598-025-02759-5

14. New Relic. (2024, September 12). AI in observability: Advancing system monitoring and analytics. *New Relic Blog.*

15. Procurri. (2025, July 15). Global OS market share 2025: Key stats, trends, and insights for mobile and desktop. https://www.procurri.com/knowledge-hub/global-os-market-share-2025-key-stats-trends-and-insights-for-mobile-and-desktop/

16. 6sense. (2025). Microsoft Windows OS - Market share, competitor insights in server and desktop OS. https://6sense.com/tech/server-and-desktop-os/microsoft-windows-os-market-share

17. StatCounter. (2025). Operating system market share worldwide. https://gs.statcounter.com/os-market-share

18. Xu, R., Xie, Z., & Chen, P. (2025). eACGM: Non-instrumented performance tracing and anomaly detection towards machine learning systems. *arXiv preprint*.

19. Levner, E. (2020). *Artificial intelligence for adaptive resource management in complex systems*. Procedia Computer Science, 176, 1234–1242.

20. Yu, S., & Wang, Z. (2024). An Empirical Study on Bugs in Rust Programming Language.

21. *International Conference on Software Quality, Reliability and Security*.

22. Jiang, R., Dong, P., Ding, Y., Wei, R., & Jiang, Z. (2023). Thetis: A Booster for Building Safer Systems Using the Rust Programming Language. *Applied Sciences*.

23. Zakopal, P., Kucera, J., Baum, F., & Bauer, J. (2024). Open-Source Internal Signal Analysis Unit for FPGA Paired With Rust Real-Time Monitor GUI. *IEEE PEMC*.

24. Sinha, S., Kalwani, P., Shah, A., & Gonsalves, J. (2025). High-Performance File Searching with Rust. *IATMSI*.

25. Minin, M. S. (2025). Enhancing Python Performance With Rust Integration. *ICIEAM*.

26. Hardin, D. S. (2023). Verification of a Rust Implementation of Knuth's Dancing Links

using ACL2. *ACL2 Workshop*.

27. Yosifovich, P., Russinovich, M., Solomon, D. A., & Ionescu, A. (2017). Windows Internals, Part 1.

28. Assi, M. J., Fahad, A., & Al-Sarray, B. (2022). Root Cause Analysis And Improvement In Windows System Based On Windows Performance Toolkit WPT. *Iraqi Journal of Science*.

APPENDIX

EXPLORER

RUST-AI_SYSTEMS_MONITOR

- src
  - analysis.rs
  - display.rs
  - gemini.rs
  - gui.rs
  - lib.rs
  - main.rs
  - metrics.rs
- target
- .env
- .gitignore M
- Cargo.lock
- Cargo.toml M
- list_models.rs

OUTLINE

analysis.rs

src > analysis.rs > ...

```rust
4   pub fn analyze_metrics(metrics: &Metrics) -> String {
11          alerts.push(format!(
12              "⚠ High RAM usage detected: {:.1}%!",
13              metrics.ram_usage_pct
14          ));
15      }
16      if metrics.disk_usage_pct > 80.0 {
17          alerts.push(format!(
18              "⚠ High Disk usage detected: {:.1}%!",
19              metrics.disk_usage_pct
20          ));
21      }
22      if metrics.network_activity > 100 * 1024 * 1024 {
23          alerts.push("⚠ High Network activity detected (>100MB)!".to_string());
24      }
25
26      if alerts.is_empty() {
27          "✅ System is stable".to_string()
28      } else {
29          alerts.join("\n")
30      }
31  }
32
```

EXPLORER

RUST-AI_SYSTEMS_MONITOR

- src
  - analysis.rs
  - display.rs
  - gemini.rs
  - gui.rs
  - lib.rs
  - main.rs
  - metrics.rs
- target
- .env
- .gitignore M
- Cargo.lock
- Cargo.toml M
- list_models.rs

OUTLINE

TIMELINE

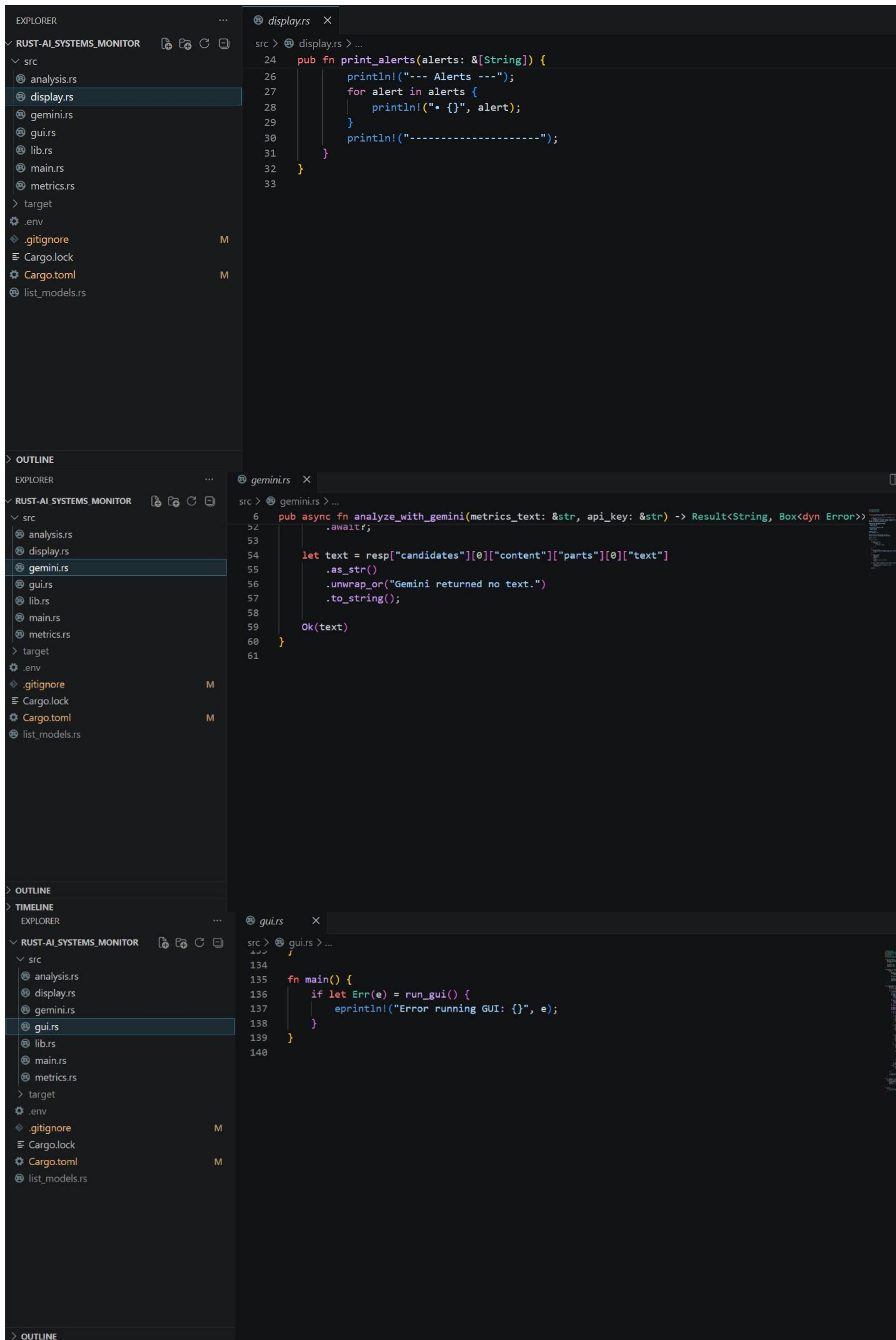
RUST DEPENDENCIES

Cargo.toml M

Cargo.toml

```toml
1   [package]
2   name = "system_monitor"
3   version = "0.1.0"
4   edition = "2021"
5   default-run = "system_monitor"
6
7   [dependencies]
8   sysinfo = "0.29"
9   humansize = "2.1"
10  chrono = "0.4"
11  csv = "1.1"
12  google-ai-rs = "0.1.2"
13  reqwest = { version = "0.12.23", features = ["json"] }
14  serde = { version = "1.0", features = ["derive"] }
15  serde_json = "1.0.143"
16  tokio = { version = "1", features = ["full"] }
17  dotenv = "0.15"N
18  dotenvy = "0.15.7"
19  eframe = "0.33.2"
20
21  [[bin]]
22  name = "gui"
23  path = "src/gui.rs"
24
25  [[bin]]
26  name = "list_models"
27  path = "list_models.rs"
```

main*+ ⊗ 0 ⚠ 0 ⊗ rust-analyzer Ln 1, Col 1 Spaces: 4 UTF-8 LF { } Plain Text


EXPLORER
RUST-AI_SYSTEMS_MONITOR
src
analysis.rs
display.rs
gemini.rs
gui.rs
lib.rs
main.rs
metrics.rs
target
.env
.gitignore M
Cargo.lock
Cargo.toml M
list_models.rs
OUTLINE
display.rs
src > display.rs > ...
24 pub fn print_alerts(alerts: &[String]) {
26 println!("--- Alerts ---");
27 for alert in alerts {
28 println!("• {}", alert);
29 }
30 println!("--------------------");
31 }
32 }
33
gemini.rs
src > gemini.rs > ...
6 pub async fn analyze_with_gemini(metrics_text: &str, api_key: &str) -> Result<String, Box<dyn Error>>
53
54 let text = resp["candidates"][0]["content"]["parts"][0]["text"]
55 .as_str()
56 .unwrap_or("Gemini returned no text.")
57 .to_string();
58
59 Ok(text)
60 }
61
TIMELINE
gui.rs
src > gui.rs > ...
134
135 fn main() {
136 if let Err(e) = run_gui() {
137 eprintln!("Error running GUI: {}", e);
138 }
139 }
140

EXPLORER

RUST-AI_SYSTEMS_MONITOR
- src
  - analysis.rs
  - display.rs
  - gemini.rs
  - gui.rs
  - lib.rs
  - main.rs
  - metrics.rs
- target
- .env
- .gitignore M
- Cargo.lock
- Cargo.toml M
- list_models.rs

OUTLINE

TIMELINE

RUST DEPENDENCIES

lib.rs

src > lib.rs > { } metrics

```rust
pub mod metrics;
pub mod display;
pub mod analysis;
pub mod gemini;

pub const VERSION: &str = "1.2.0";
```

main*+ ⊗ 0 ⚠ 0 ⊗ rust-analyzer Ln 1, Col 1 Spaces: 4 UTF-8 CRLF { } Rust

EXPLORER

RUST-AI_SYSTEMS_MONITOR

- src
  - analysis.rs
  - display.rs
  - gemini.rs
  - gui.rs
  - lib.rs
  - main.rs
  - metrics.rs
- target
- .env
- .gitignore M
- Cargo.lock
- Cargo.toml M
- list_models.rs

OUTLINE

TIMELINE

RUST DEPENDENCIES

main.rs

src > main.rs > ...

```
20   async fn main() {
107              _ = sleep(Duration::from_secs(5)) => {}
108          }
109      }
110  }
```

EXPLORER

RUST-AI_SYSTEMS_MONITOR

- src
  - analysis.rs
  - display.rs
  - gemini.rs
  - gui.rs
  - lib.rs
  - main.rs
  - metrics.rs
- target
- .env
- .gitignore M
- Cargo.lock
- Cargo.toml M
- list_models.rs

OUTLINE

TIMELINE

RUST DEPENDENCIES

metrics.rs

src > metrics.rs > ...

```
1   use sysinfo::{System, SystemExt, CpuExt, DiskExt, NetworksExt, NetworkExt};
2   use serde::Serialize;
3
4   #[derive(Debug, Serialize)]
5   pub struct Metrics {
6       pub cpu_usage: f32,          // CPU usage (%)
7       pub ram_usage_bytes: u64,    // RAM used (bytes)
8       pub ram_usage_pct: f32,      // RAM usage (%)
9       pub disk_usage_bytes: u64,   // Disk used (bytes)
10      pub disk_usage_pct: f32,     // Disk usage (%)
11      pub network_activity: u64,   // Network traffic (bytes)
12  }
13
14  pub fn collect_metrics() -> Metrics {
15      let mut sys = System::new_all();
16      sys.refresh_all();
17
18      // CPU %
19      let cpu_usage = sys.global_cpu_info().cpu_usage();
20
21      // RAM (bytes + %)
22      let ram_usage_bytes = sys.used_memory();
23      let total_ram = sys.total_memory();
24      let ram_usage_pct = if total_ram > 0 {
25          (ram_usage_bytes as f32 / total_ram as f32) * 100.0
26      } else {
27          0.0
28      };
29
```