Final Report: LLM Inventory Management System

1. Introduction

This project is a robust, multi-faceted system designed to analyze, monitor, and optimize server performance, with a focus on high-frequency trading (HFT) applications. The system provides detailed insights into server hardware and software configurations, monitors system health and performance in real-time, offers performance optimization recommendations, and suggests hardware upgrades tailored to user budgets. A standout feature is its integration of a chatbot powered by Grok (an AI model from xAI) and agentic Retrieval-Augmented Generation (RAG), for intuitive natural language interaction. The primary goals are to enhance server reliability, optimize performance for latency-sensitive environments, and simplify server management for users.

The project comprises several Python scripts (`collect_data.py`, `rag_implement_v3.py`, `m_monitor_system_analyzer.py`, `upgrade_recommender.py`, `server.py`, `upgrade_recommender_2.py`, `performance_optimizer.py`, `m_monitor_system.py`) and a frontend interface (`index.html`), working together to deliver a comprehensive server management solution.

2. System Components

The system is modular, with each component serving a distinct role:

2.1 Data Collection (`collect_data.py`)

- **Purpose**: Gathers detailed system metrics using Linux tools like `lscpu`, `top`, `lspci`, `sensors`, and `dmidecode`.
- **Output**: Generates `metrics_config.json` (configuration file) and `system_info.txt` (raw metrics data).
- **Features**: Dynamically detects hardware components (e.g., NICs, GPUs, disks) and verifies tool availability, suggesting installation commands if tools are missing. Users can define custom metrics, which integrate seamlessly into the Monitor and Performance tabs, enhancing adaptability for specific server needs.

2.2 Query Handling and RAG (`rag_implement_v3.py`)

- **Purpose**: Implements a chatbot interface using Grok and RAG to answer user queries.
- **Functionality**: Categorizes queries into system information, monitoring, or upgrades, retrieving relevant data from `system_info.txt` using sentence embeddings and a vector store (Chroma).

2.3 Monitoring (`m_monitor_system_analyzer.py` and `m_monitor_system.py`)

- **Analyzer (`m_monitor_system_analyzer.py`)**: Fetches safe operational thresholds for hardware components using Grok, storing them in a dictionary.
- **Monitor (`m_monitor_system.py`)**: Continuously collects and displays metrics based on `monitor_settings.json`, logs data, and alerts on threshold violations. Filters entries by the type field, selecting only those with dynamic_single, which output a single numerical value

2.4 Upgrade Recommendations (`upgrade_recommender.py` and `upgrade_recommender_2.py`)

- **Purpose**: Analyzes `system_info.txt` to suggest hardware upgrades.
- **Features**: Uses Grok to provide recommendations within a budget, caching results to avoid redundant API calls. `upgrade_recommender_2.py` is an enhanced version with chatbot integration.

2.5 Performance Optimization (`performance_optimizer.py`)

- **Purpose**: Optimizes server performance for HFT, focusing on latency and throughput.
- **Features**: Analyzes metrics, including user-defined ones, and generates tuning scripts (e.g., `tune_system.sh`), and shows pre-tuning performance to serve as a baseline for comparison after re-running the analysis.

2.6 Backend Server (`server.py`)

- **Purpose**: A Flask application serving as the system's backend.
- **Features**: Handles API requests for data collection, monitoring, performance optimization, upgrades, and chatbot interactions, with SocketIO for real-time updates.

2.7 Frontend Interface (`index.html`)

- **Purpose**: Provides a user-friendly dashboard.
- **Features**: Enables interaction with system features (e.g., viewing metrics, starting monitoring, applying optimizations). While partially implemented, the frontend offers functional controls, with plans for enhanced visualizations and user controls in future development.

3. Implementation Details

3.1 Data Collection

- **Mechanism**: `collect_data.py` uses subprocesses to execute system commands, dynamically generating metrics based on discovered components (e.g., NICs, sensors). It validates metric outputs and handles missing tools gracefully.
- **Integration**: Outputs feed into other components for analysis and monitoring.

3.2 Query Handling

- **RAG Process**: `rag_implement_v3.py` splits `system_info.txt` into chunks, embeds them using `SentenceTransformer`, and retrieves relevant data for Grok to process user queries.
- **Routing**: Queries are categorized (e.g., system info, monitoring) and routed to appropriate scripts (e.g., `m_monitor_system_analyzer.py` for monitoring).

3.3 Monitoring

- **Thresholds**: `m_monitor_system_analyzer.py` queries Grok for hardware-specific thresholds, ensuring safe operation limits.
- **Real-Time**: `m_monitor_system.py` runs a background thread to collect metrics at configurable intervals, displaying them in a color-coded table with downloadable logs for post-analysis. Users can enable ping host checks and integrate custom metrics, with Grok-assisted or manual threshold settings.

3.4 Upgrade Recommendations

- **Analysis**: Both `upgrade_recommender.py` and `upgrade_recommender_2.py` parse `system_info.txt`, summarize key metrics (e.g., CPU, memory), and use Grok to suggest upgrades.
- **Interactivity**: The Upgrade tab's dedicated Grok Q&A allows follow-up questions (e.g., "Which upgrade is most urgent?") for concise, data-driven guidance.
- **Caching**: Results are cached in `upgrade_cache.json` to optimize performance.

3.5 Performance Optimization

- **Focus**: `performance_optimizer.py` targets HFT-relevant metrics (e.g., NIC ring buffers, CPU affinity), generating bash scripts for tuning.
- **Validation**: Collects pre- and post-tuning metrics to measure impact.
- **Process**: Users select profiles or metric categories, including custom metrics. The system provides commands for manual execution, a metrics tab for transparency, and downloadable pre/post-tuning reports to measure impact.

3.6 Backend and Frontend

- **Server**: `server.py` uses Flask for RESTful APIs and SocketIO for real-time updates, integrating all components.
- **Frontend**: `index.html` provides a control panel, though its implementation is partial, relying on backend APIs for functionality.

3.7 AI Integration

- **Grok**: Used across components for natural language processing, threshold generation, and recommendation analysis, leveraging the xAI API.

4. Problems Addressed

This system tackles several critical challenges in server management:

4.1 System Analysis and Monitoring

- **Problem**: Lack of comprehensive, real-time server insights.
- **Solution**: Provides detailed hardware/software analysis and continuous monitoring with threshold-based alerts, enabling proactive maintenance.

4.2 Performance Optimization for HFT

- **Problem**: Suboptimal performance in latency-sensitive applications.
- **Solution**: Analyzes and tunes system parameters (e.g., IRQ affinity, NUMA binding), delivering measurable improvements for HFT workloads.

4.3 Hardware Upgrade Planning

- **Problem**: Difficulty in identifying cost-effective upgrade options.
- **Solution**: Offers data-driven upgrade recommendations within budget constraints, enhancing system longevity and performance.

4.4 User-Friendly Interaction

- **Problem**: Technical complexity barriers for non-expert users.
- **Solution**: The Grok-powered chatbot allows natural language queries, simplifying interaction and reducing the learning curve.

4.5 Real-Time Monitoring and Alerts

- **Problem**: Delayed detection of performance issues or failures.
- **Solution**: Real-time metric collection and violation alerts ensure timely responses to potential problems.

5. Demo Video

A video demonstrating how the project works: **Demo_Video_Group 19**

6. Future Work

Future enhancements include integrating `performance_optimizer.py` into the chatbot for natural language-driven optimization queries. Leveraging raw data logs and customizable metrics could enable advanced features like predictive maintenance models or automated anomaly detection. Expanding the frontend with enhanced visualizations and controls will further improve usability, making the system a robust platform for continuous server improvement.

7. Conclusion

This server analysis and optimization system is a powerful tool for system administrators and engineers, particularly those managing HFT infrastructure. By integrating detailed analysis, real-time monitoring, AI-driven recommendations, and an accessible interface, it streamlines server management and enhances performance and reliability. Its modular design allows for future enhancements, such as expanded monitoring capabilities or additional optimization profiles, making it a versatile solution for critical server environments.