

# Raspberry Pi Monitoring Bot: Technical Documentation

## 1. Project Overview and Architectural Summary

### 1.1. Introduction to the Raspberry Pi Monitoring Bot

The Raspberry Pi Monitoring Bot is a specialized Python-based application designed to provide comprehensive, remote oversight of a Raspberry Pi's operational status. The system's primary function is to gather and present real-time and historical data on critical system resources and network connectivity through an intuitive Telegram interface. This allows system administrators or advanced users to perform remote diagnostics, initiate maintenance tasks, and receive automated alerts and performance reports without direct physical access to the device.

The bot's capabilities are segmented into several key areas: real-time system status checks (CPU, RAM, storage, and temperature), network connectivity analysis (ping latency, jitter, and packet loss), IP configuration retrieval, and historical report generation. It also incorporates a secure "Expert Mode" that permits the execution of a predefined set of whitelisted shell commands for advanced diagnostics. The bot's core functionality is orchestrated through a central script that integrates with the Telegram Bot API, while its data collection and reporting mechanisms operate through independent, decoupled modules.

### 1.2. High-Level System Architecture

The bot's architecture is founded on a distributed, multi-process design, which enhances its robustness and resilience. The system is composed of several distinct, yet interconnected,

components that operate in concert to deliver the bot's functionality. The core orchestrator is `bot.py`, which serves as the central hub for all user interactions.<sup>1</sup> It listens for incoming commands and callback queries from the Telegram API, processes them, and routes requests to the appropriate internal modules.

Data collection is handled by two dedicated background processes: `system_monitor.py` and `module1.py`.<sup>1</sup> These scripts are designed to run continuously and independently of the main bot logic.

`system_monitor.py` is responsible for collecting system resource metrics and logging them, while `module1.py` focuses on network connectivity metrics. This separation of concerns ensures that a user request for a report or a real-time status update does not block the continuous data collection process.

All collected system and network metrics are stored in a central SQLite database, `monitoring_data.db`, which acts as the single source of truth for historical data.<sup>1</sup> The

`report_generator.py` module, which is called by the main bot script, accesses this database to query historical data and generate analytical PDF reports using `pandas` and `matplotlib`.<sup>1</sup> Security is addressed by the

`command_filter.py` module, which provides a critical layer of validation for any shell commands executed via the bot's "Expert Mode".<sup>1</sup> This architectural model of separate, single-purpose components communicating through a central database is a scalable and maintainable approach for a project of this scope.

### 1.3. End-to-End Data Flow Analysis

The system's operation can be understood by analyzing its two primary data flows: the asynchronous data collection loop and the user-triggered report generation pipeline.

#### Flow 1: Asynchronous Data Collection

The foundation of the monitoring bot is its continuous, background data collection. This flow is managed by two separate scripts running in perpetual loops.


1. The `system_monitor.py` script, when executed, enters a while True loop that runs every 60 seconds.<sup>1</sup>
2. Inside this loop, it calls the `log_system_metrics()` function, which uses the `psutil` library to retrieve CPU, RAM, and storage metrics and `vcgencmd` to get the Raspberry Pi's CPU temperature.<sup>1</sup>

3. The collected metrics are then passed to the `save_to_db()` function, which persists them as a new row in the `system_resources` table of the `monitoring_data.db` database.<sup>1</sup>
4. Similarly, `module1.py` runs in a separate while True loop that executes every 180 seconds.<sup>1</sup>
5. It iterates through a set of hardcoded hosts (e.g., Google DNS, Cloudflare DNS, and a local gateway) and uses the `ping_host()` function to measure latency, jitter, and packet loss.<sup>1</sup>
6. The results of the network checks are then saved to the `network_logs` table via its own `save_to_db()` function.<sup>1</sup>

This architectural design is a deliberate choice to decouple the data collection processes from the user-facing bot interface. This prevents any potential delays or resource-intensive tasks, such as a prolonged ping, from negatively impacting the bot's responsiveness to user commands. The primary consequence of this design is that the database holds a historical record of system state at fixed intervals rather than a perfectly real-time stream. This is a critical consideration when interpreting the data and reports, as they represent snapshots in time rather than a continuous, live view of the system's performance.

## Flow 2: User-Triggered Report Generation

This flow begins when a user initiates a request for historical data via the Telegram interface.

1. A user sends the `/start` command, which triggers the welcome handler in `bot_clean.py`.<sup>1</sup> This handler validates the user's chat ID and presents an inline keyboard with various options, including " Report Generation".<sup>1</sup>
2. The user selects this option, sending a `callback_data="report_menu"` query to the bot. This is processed by the `handle_callbacks` function, which routes the request to the `show_report_menu()` function.<sup>1</sup>
3. `show_report_menu()` presents another inline keyboard with predefined time ranges (e.g., "Last 24 Hours," "Today").<sup>1</sup>
4. When the user selects a time range, a new callback query (e.g., "report\_last\_24") is sent and handled by `generate_and_send_report`.<sup>1</sup>
5. The `generate_and_send_report` function invokes the `generate_report()` function from `report_generator.py`, passing the selected time range as a parameter.<sup>1</sup>
6. The `generate_report()` function constructs SQL queries to fetch the relevant time-series data from the `system_resources` and `network_logs` tables using `pandas`.<sup>1</sup>
7. The retrieved data is then used to generate a series of PDF plots using `matplotlib`, which are compiled into a single PDF report file in the `REPORT_DIR`.<sup>1</sup>
8. Finally, the main bot script uses `bot.send_document()` to send the newly created PDF file to the user via Telegram.<sup>1</sup> This end-to-end process effectively transforms raw, logged data into a consumable, visual report.

## 2. Module-by-Module Technical Analysis

### 2.1. Core Bot Logic: bot.py

The bot.py script serves as the control center for the entire application. It is built on the pyTelegramBotAPI library and is responsible for managing all user interactions and delegating tasks to other modules. The bot utilizes a simple but effective access control mechanism through the is\_allowed\_user() function, which checks if a user's chat ID is present in a hardcoded whitelist (ALLOWED\_CHAT\_ID).<sup>1</sup> This method is suitable for a private bot with a limited user base but would require a more scalable, persistent solution (e.g., a database table) for a larger deployment.

The script's main loop, bot.infinity\_polling(), is a synchronous, event-driven mechanism that blocks until it receives a new message or callback query from the Telegram API.<sup>1</sup> This ensures that the bot remains responsive to user input at all times. In addition to handling user requests, the script also manages an automated daily reporting feature. It achieves this by launching the

auto\_daily\_report() function in a separate daemon thread at startup.<sup>1</sup> This design allows the bot to perform a background task—generating and sending yesterday's report—without interfering with its primary function of handling user queries.

A key architectural nuance exists in how the bot retrieves information from its different modules. When a user requests system status via the show\_system\_status() function, the bot calls get\_system\_status() in system\_monitor.py. This function uses psutil to pull live, real-time metrics (CPU usage, RAM, etc.) at the moment of the request.<sup>1</sup> In contrast, when a user requests network status via

show\_network\_status(), the bot calls get\_network\_status() in module1.py, which retrieves the *most recent* logged data from the database using a LIMIT 1 SQL query.<sup>1</sup> This hybrid model—fetching live data for fast, lightweight system checks and serving cached data for potentially slower network checks—is a pragmatic design choice that prioritizes bot responsiveness.

### 2.2. System Resource Monitoring: system\_monitor.py

This module is responsible for the continuous collection of system metrics and the storage of this data for historical analysis. It leverages the psutil library to programmatically access system information, including CPU usage and virtual memory statistics.<sup>1</sup> For Raspberry Pi-specific hardware monitoring, it uses

`os.popen()` to execute the `vcgencmd measure_temp` shell command and parse its output to retrieve the CPU temperature.<sup>1</sup> The module also provides functions to retrieve public and private IP addresses using external services (

requests) and local network interfaces (psutil, socket).<sup>1</sup>

The `system_monitor.py` script incorporates an internal alerting system that checks the collected metrics against predefined warning and critical thresholds.<sup>1</sup> When a metric exceeds a critical threshold, an alert message is generated and appended to a global, in-memory list called

`stored_alerts`. The `get_stored_alerts()` function retrieves the contents of this list and, immediately after, clears it.<sup>1</sup> This design has significant limitations. Because the alert list is stored in memory, it is not persistent across restarts of the bot, and any critical alerts would be lost. Furthermore, since the list is cleared upon retrieval, an alert retrieved by one user would no longer be visible to any other allowed users, hindering collaborative monitoring and creating an unreliable alert history. For a robust monitoring system, alerts should be logged to the persistent SQLite database, with timestamps and an acknowledgment status, to ensure an auditable and comprehensive alert history.

## 2.3. Network Connectivity Monitoring: module1.py

The `module1.py` script is dedicated to monitoring external network connectivity. It employs `subprocess.run()` to execute a standard ping command to a set of predefined hosts.<sup>1</sup> The script then uses regular expressions (

re) to parse the output of the ping command to extract key metrics such as average latency, jitter, and packet loss.<sup>1</sup> It is important to note that the jitter calculation is simplified to the difference between the maximum and minimum latency values. These collected metrics are then stored in the

`network_logs` table of the database.<sup>1</sup>

A minor architectural inconsistency is present in this module: the `create_table()` function, which ensures the `network_logs` table exists, is also defined and called in the `system_monitor.py` script for its own table.<sup>1</sup> While this duplication is functionally harmless, it violates the principle of writing maintainable and non-redundant code. In a more complex system, this approach could lead to synchronization issues and an increased risk of schema inconsistencies if one module's table definition is updated but the other is not.

## 2.4. Report Generation and Analytics: `report_generator.py`

This module is the analytical core of the system, responsible for transforming raw database data into human-readable performance reports. It relies on the `pandas` library to perform efficient data retrieval and manipulation through SQL queries against the SQLite database.<sup>1</sup> The retrieved data frames are then used by

`matplotlib` to generate a series of time-series plots, which are compiled into a multi-page PDF document using `PdfPages`.<sup>1</sup> The module's functionality includes generating a summary page with key statistics (e.g., uptime percentage and average resource utilization) and detailed plots for each metric (latency, CPU usage, etc.) over a specified time range.

A critical maintenance consideration for a long-running system on a constrained device like a Raspberry Pi is data management. The `purge_old_reports()` function in this module is designed to delete PDF reports older than 24 hours from the filesystem to free up storage space.<sup>1</sup> However, this function does not address the indefinite growth of the underlying SQLite database. The

`system_resources` and `network_logs` tables will continue to grow with a new entry every 60 and 180 seconds, respectively, without a corresponding data pruning mechanism. If left unaddressed, this will eventually lead to the database file consuming all available storage on the device, causing system instability and operational failure.

## 2.5. Security and Command Filtering: `command_filter.py`

The `command_filter.py` module is a dedicated security component that enforces a crucial policy for the bot's "Expert Mode".<sup>1</sup> When a user requests to execute a shell command, the

`execute_shell_command()` handler in the main bot script first validates the command using `is_safe_command()` from this module.<sup>1</sup> The validation mechanism is based on a simple

whitelist approach, which checks if the first word of the user's command is present in the `SAFE_COMMANDS` list.<sup>1</sup>

This whitelist is a fundamental security barrier, but its integrity is paramount. The core bot script uses `subprocess.run(cmd, shell=True)` to execute the user's command.<sup>1</sup> The use of

`shell=True` can be a significant security risk, as it allows for command chaining and injection if user input is not properly sanitized. In this case, the security of the entire "Expert Mode" is entirely dependent on the strictness and completeness of the `SAFE_COMMANDS` list and the one-word check performed by `is_safe_command`. While this simple check is effective for the current list of commands, it is a fragile security model. Any expansion of the whitelist would require a thorough security review to prevent the addition of commands that could be exploited to compromise the system.

## **3. Database Schema and Data Management**

### **3.1. Database Role and Architecture**

The SQLite database (`monitoring_data.db`) is the central repository for all logged system and network metrics. Its role is twofold: it provides a historical record of performance for long-term analysis, and it acts as the primary data source for the report generation module. The choice of SQLite is well-suited for this application due to its serverless, file-based nature. This eliminates the need for a separate database server process, making the system lightweight and ideal for deployment on a single-board computer like the Raspberry Pi. The database is accessed by the various Python scripts using the built-in `sqlite3` library.<sup>1</sup>

### **3.2. Database Schema Definition**

The database consists of two primary tables designed to store system and network data, respectively. A clear understanding of the schema is essential for querying the data and ensuring its integrity. The following table provides a formal definition of the columns, their

data types, and their purpose.

Table Name	Column Name	Data Type	Description	Source
system_resources	timestamp	TEXT	The timestamp of the log entry.	1
	cpu_temp	REAL	CPU temperature in degrees Celsius.	1
	cpu_usage	REAL	Percentage of CPU usage.	1
	ram_usage	REAL	Percentage of RAM usage.	1
	storage_usage	REAL	Percentage of storage space used.	1
network_logs	timestamp	TEXT	The timestamp of the network check.	1
	host	TEXT	The name or IP of the host being pinged.	1
	latency	REAL	Average network latency in milliseconds.	1
	jitter	REAL	The measured jitter in	1



			milliseconds.	
	packet_loss	REAL	Percentage of packets lost.	<sup>1</sup>
	status	TEXT	The status of the host ("UP" or "DOWN").	<sup>1</sup>

The current schema is well-designed for its intended purpose. However, its rigid structure may limit future extensibility. For instance, adding a new metric, such as GPU temperature or a specific service's uptime, would require a schema migration. A more extensible design might use a key-value pair system for metrics, but the current schema is efficient and perfectly adequate for the bot's present scope.

## 4. Configuration and Deployment Guide

### 4.1. Core Configuration Parameters

Effective deployment and maintenance require a clear understanding of the bot's configuration parameters. These variables, which control key aspects of the bot's behavior, are currently scattered across different script files. A significant maintenance challenge arises from the fact that `DB_PATH`, for example, is hardcoded in three separate locations: `module1.py`, `report_generator.py`, and `system_monitor.py`.<sup>1</sup> In contrast, the

`BOT_TOKEN` is correctly loaded from an external `.env` file.<sup>1</sup> This inconsistency violates the principle of a "single source of truth" and makes modifying the database path a manual and error-prone process. Centralizing all such configuration variables into a single

`.env` file would greatly enhance the system's maintainability and ease of deployment. The following table details the most critical configuration parameters.

Parameter Name	Script Location	Purpose	Source
----------------	-----------------	---------	--------

BOT_TOKEN	bot_clean.py	The unique token for the Telegram bot, loaded from an external .env file.	1
ALLOWED_CHAT_ID	bot_clean.py	A set of hardcoded chat IDs that are granted access to the bot's functions.	1
UPTIME_ALERT_THRESHOLD	bot_clean.py	The percentage threshold for uptime alerts.	1
DB_PATH	module1.py, report_generator.py, system_monitor.py	The absolute filesystem path to the SQLite database file.	1
REPORT_DIR	report_generator.py	The directory where generated PDF reports are stored.	1
WARNING_*/CRITICAL_*	system_monitor.py	A set of integer thresholds for various system metrics (CPU temp, RAM usage, storage usage) that trigger alerts.	1
SAFE_COMMANDS	command_filter.py	A whitelist of shell commands permitted for execution via "Expert Mode."	1

## 4.2. Deployment and Maintenance

The bot is not a single executable but a collection of distinct processes. For a full deployment, `bot_clean.py`, `system_monitor.py`, and `module1.py` must be initiated and maintained as separate, long-running processes.<sup>1</sup> A process manager like

`systemd` or tools like `nohup` are highly recommended to ensure that these scripts start automatically on system boot and are restarted in the event of a failure.

During the analysis of the `system_monitor.py` script, a function named `set_alert_threshold()` was identified.<sup>1</sup> This function is designed to dynamically modify the alert thresholds but is never called within the provided codebase. This suggests a planned but currently unimplemented feature. The integration of this function into the bot's user interface would represent a valuable future enhancement, allowing administrators to configure alert parameters remotely without modifying the source code.

## 5. Conclusions and Recommendations

The Raspberry Pi Monitoring Bot is a well-designed and highly functional application that effectively addresses the need for remote system oversight. Its core strengths lie in its distributed architecture, which separates data collection from user interaction, and its reliance on robust, open-source libraries for monitoring (`psutil`), analysis (`pandas`, `matplotlib`), and communication (`telebot`). The use of a persistent database to store metrics is a sound design decision that enables the generation of valuable historical reports.

However, the analysis of the codebase reveals several key areas for improvement to ensure the long-term reliability and maintainability of the system. Based on these findings, the following recommendations are provided:

1. **Implement a Database Pruning Mechanism:** The current system lacks a function to delete old records from the SQLite database, which will cause the database file to grow indefinitely and eventually consume all available storage. It is highly recommended to add a function to `report_generator.py` that, similar to `purge_old_reports()`, periodically removes database entries older than a specified retention period (e.g., 30 days).
2. **Refactor the Alerting System:** The in-memory, non-persistent alert list is a critical weakness. It is recommended to refactor the alert mechanism to store all critical alerts as records in a new table within the SQLite database. This would provide a persistent, auditable, and reliable history of all system alerts, accessible to all authorized users and preserved across system restarts.

3. **Centralize Configuration:** The current practice of hardcoding configuration paths (DB\_PATH, REPORT\_DIR) across multiple files is an anti-pattern. All configurable parameters should be consolidated into a single .env file, loaded by all modules at startup. This single source of truth would significantly simplify future maintenance and deployment.
4. **Enhance Expert Mode Security:** While the command whitelist is an effective security measure, the reliance on shell=True for subprocess execution makes the system vulnerable if the whitelist is ever compromised. Any future additions to the SAFE\_COMMANDS list should be carefully vetted. As an alternative, consider a more restrictive execution model, such as one that avoids shell=True and passes the command and its arguments as a list to subprocess.run().

## Works cited

1. bot.txt