Software Design Document (SDD)

ARCHITECTURAL STYLE

The software is designed in a monolithic architectural style, implemented in C# using the .NET Framework 4.8 with a WinForms-based graphical interface. This design fits very well for a standalone Windows desktop application, as it encapsulates all core components such as inventory capture, difference analysis, reporting, and configuration management within a single cohesive system. By following a monolithic structure, the application reduces the complexity related to deployment and maintenance while guaranteeing performance and compatibility across supported Windows environments. This approach is also effective in directly utilizing Windows APIs without the additional overhead of distributed services or cross-platform dependencies for reliable file system and registry monitoring. The architecture focuses on simplicity, portability, and user accessibility based on the primary goal of the project: offering a transparent, dependable monitoring tool for developers, system administrators, and security analysts.

The layered architecture diagram in ***Figure 1*** below illustrates how the major components of WinChangeMonitor interact to form a cohesive monolithic system. The design follows a four-layer structure made up of the Presentation Layer, Application Services Layer, Domain Layer, and Infrastructure Layer. The Presentation Layer, implemented with WinForms, serves as the user interface through which users initiate scans, monitor progress, and view results. The Application Services Layer coordinates the system's core operations, including inventory orchestration, difference analysis, and report generation. The Domain Layer defines the primary models and policies for representing file, registry, and service data and the logic used in comparing system states. Finally, the Infrastructure Layer interacts directly with the Windows APIs for file system, registry, and service access and manages data persistence through JSON, XML, or SQL storage formats. This layered design promotes separation of concerns, with each layer focused on a specific aspect of functionality.

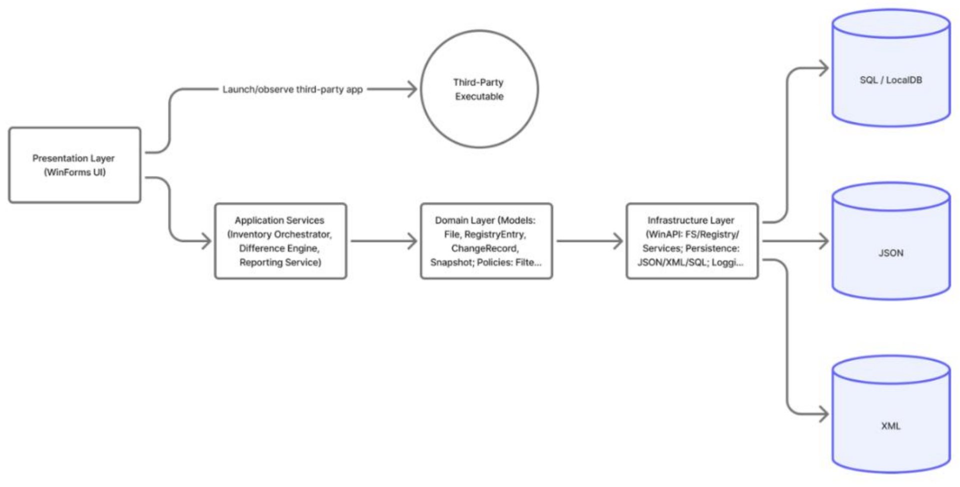
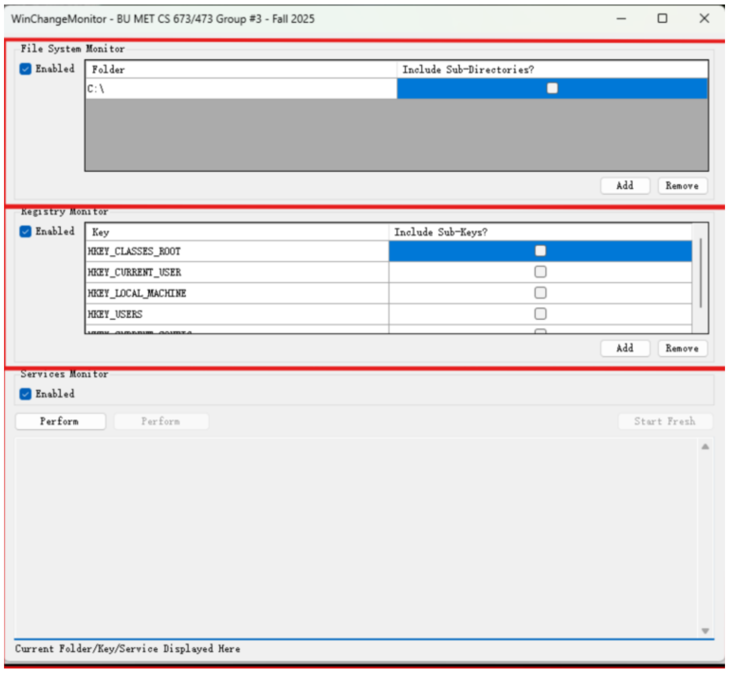

Presentation Layer (WinForms UI)
Launch/observe third-party app
Third-Party Executable
Application Services (Inventory Orchestrator, Difference Engine, Reporting Service)
Domain Layer (Models: File, RegistryEntry, ChangeRecord, Snapshot; Policies: Filte...
Infrastructure Layer (WinAPI: FS/Registry/ Services; Persistence: JSON/XML/SQL; Loggi...
SQL / LocalDB
JSON
XML

Figure 1: Layered Architecture Diagram for WinChangeMonitor

HIGH-LEVEL DESIGN

User Interface Overview

The WinChangeMonitor user interface is designed to provide a clear and modular layout for monitoring file system, registry, and service changes on Windows systems. The User Interface (UI) will be presented as ***Figure 2*** below. The user interface has 3 major parts, including File System Monitor, Registry Monitor and output console. The top section, labeled File System Monitor, allows users to specify folders for monitoring by entering directory paths and optionally including subdirectories. Each folder entry appears in a grid view, with checkboxes to control recursion and buttons to add or remove monitored locations. Below it, the Registry Monitor section offers similar functionality, enabling the user to select Windows registry hives or specific keys to monitor, with an option to include sub-keys. Each section includes an Enabled checkbox to independently activate or deactivate monitoring features. At the very bottom of the of UI is the output console. User will see an overview report of what has been change compared to last inventory. By clicking Start Fresh, user can start a new inventory and have it ready for next comparison.



Folder	Include Sub-Directories?
C:\	☐

Key	Include Sub-Keys?
HKEY_CLASSES_ROOT	☐
HKEY_CURRENT_USER	☐
HKEY_LOCAL_MACHINE	☐
HKEY_USERS	☐

Figure 2: UI Design for WinChangeMonitor

Layered Architecture

The application is structured as a layered monolithic system with four principal modules: the presentation layer, application services layer, domain layer, and infrastructure layer. The presentation layer is implemented in WinForms and includes the user interface for baseline and post-execution scans, displaying progress, and showing visual reports; it performs user interactions with the system, such as target directory selection, starting the scan, and exporting reports. The application services layer facilitates system workflow via the Inventory Orchestrator, Difference Engine, and Reporting Service, maintaining core operations of capturing inventories, comparing pre- and post-execution states, and generating human-readable reports. The domain layer describes the system's core logic and data models, which include file and registry entry representations, record of changes, and inventory snapshots. It also enforces rules around filtering and anonymizing sensitive data. The infrastructure layer executes these functions through low-level interaction with the Windows APIs and file-based persistence using a binary storage format for performance and security. Logging services provide traceability by recording key system events, such as inventory execution, exports, and configuration changes. All these layers create a cohesive architecture that fosters maintainability, readability, and scalability in one deployable application.

RUNTIME BEHAVIOR

At runtime, the application follows a clear and sequential workflow that captures system changes introduced by third-party programs. Upon the user's request to create a baseline inventory, the Inventory Orchestrator calls the File and Registry Scanners; these collect information about files, directories, and registry keys, respectively, according to user configuration. Once a baseline is stored, the user executes the external software under observation. Following its execution, the application executes a second inventory that captures the current state of the system. The Difference Engine then compares both inventories for added, modified, or deleted items and summarizes these changes for reporting. The Reporting Service creates a structured HTML report and optionally exports data in PDF or JSON format for documentation or auditing. All long scanning operations run in the background throughout this process, while the WinForms interface keeps the progress indicators updated in real time. If the system is rebooted between scans, the application preserves the baseline inventory to maintain consistency. This runtime sequence ensures reliability, repeatability, and usability, providing users with accurate and clear insights into how third-party software alters their systems.

State diagrams show how the WinChangeMonitor application changes its state in response to user actions and to system events. A state diagram provides a dynamic view of the system, illustrating how processes such as initialization, inventory management, and report generation are sequenced from state to state. By modeling these transitions, it makes clear how an application maintains logical control over its operations-from startup and configuration through pre- and post-installation inventories to the creation of the final report-so that each component behaves in a predictable and consistent manner throughout execution. The state diagrams used in this project are described below.

- ***Application Initialization State Diagram*****:** This diagram models the system's startup and initialization phase. It begins with disabling controls and showing a splash screen, then checks whether an initial inventory exists on disk. Depending on that condition, the application either loads existing inventory data into memory or enables the controls to start a new monitoring session. It also determines which monitoring features (File System, Registry, Services) were previously tracked and reactivates them accordingly. Once initialization is complete, the splash screen is hidden, and control transitions to the main user interaction state. This ensures a consistent and recoverable startup process that adapts whether or not prior data exists.

- ***User Interaction and Inventory Management State Diagram:*** This state diagram captures the core user-driven workflow for performing pre-installation and post-installation inventories. It starts when the user toggles system monitors (File System, Registry, or Services), which enables or disables corresponding customization controls. From there, the user can perform a Pre-Install inventory, which scans tracked files, folders, and registry keys, saving results to disk. After running the third-party application, the Post-Install inventory phase is executed. This step initializes "New" and "Modified" structures and compares tracked entities across inventories to identify additions, modifications, or deletions. Once these steps are complete, the system transitions to the report generation process. This state machine effectively manages how the user interacts with scanning controls, ensuring proper sequencing and data consistency between scans.

- ***Report Generation State Diagram:*** This diagram explains the report creation process that occurs after the difference analysis is complete. It starts when the user initiates report generation, processing the New, Modified, and Deleted structures for files, registry keys, and services. The system compiles each of these sections into a single comprehensive report. After all content has been added, the completed report is opened in the user's default browser for viewing. This diagram highlights the sequential logic and data aggregation steps used to produce the final HTML output.

NOTE: ***Figures 3, 4, and 5*** in the appendix correspond to the respective state diagrams described above.

DETAILED SYSTEM DESIGN

Module Overview

The classes in the WinChangeMonitor system represent the internal structure of the application and define how configuration data, inventory records, and difference analysis are handled across different monitoring domains. Each class encapsulates a specific responsibility within the software, contributing to the overall workflow of capturing system states, comparing results, and presenting findings to the user. The model is designed with modularity and maintainability in mind, separating user interface logic from the underlying business rules and data-handling components. By organizing functionality into logical modules, the system ensures that updates or extensions in one area, such as registry or service monitoring, can occur without affecting other parts of the program.

At a high level, the design is composed of five interconnected modules: file system monitoring, registry monitoring, service monitoring, data persistence, and reporting. Each of these modules contains its own set of classes responsible for managing the data relevant to its domain. For example, the file system monitoring module handles the tracking of directories and files, while the registry monitoring module manages registry keys and values. The service monitoring module focuses on capturing system service configurations and operational states. The persistence module handles storage, serialization, and recovery of configuration and inventory data, ensuring continuity between sessions. Finally, the reporting module compiles all results into structured and readable formats for user review. Together, these modules form a cohesive architecture that supports the project's goals of transparency, usability, and efficient change detection across Windows environments.

Each module interacts with others through well-defined dependencies that maintain the integrity of data flow while reducing coupling between components. The table below summarizes the primary modules, their key classes, core responsibilities, and the major dependencies that support their functions within the overall architecture of the WinChangeMonitor.

Module	Primary Classes	Responsibilities	Dependencies
File System Monitoring	FileSystemSettings, TrackedFolder, FileSystemEntryInfo	Manages the tracking of selected folders and subfolders, performs baseline and post-installation inventories, and identifies new,	Depends on Windows File APIs for system access and MessagePack for

		modified, or deleted files within the monitored scope.	serialization and persistence.
Registry Monitoring	RegistrySettings, TrackedKey, RegistryEntryInfo	Handles the scanning of user-defined registry keys, records values during pre- and post-installation inventories, and detects changes in registry entries.	Depends on Windows Registry APIs, serialization utilities, and user configuration settings.
Service Monitoring	ServicesSettings, ServiceInfo	Captures and compares Windows service states and configurations, recording details such as service type, dependencies, and startup modes.	Depends on Windows Service Control Manager APIs for status retrieval and system metadata.
Persistence Layer	RetainedSettings, CommonInfo, MessagePackObject	Manages storage, loading, and serialization of system configurations and inventories. Ensures data consistency between user sessions and provides save, load, and delete operations.	Depends on MessagePack for efficient binary serialization and .NET file I/O libraries for local storage.
Reporting	RetainedSettings (methods for report generation)	Aggregates collected differences across all monitored domains and generates structured HTML reports. Optional export in PDF or JSON formats for documentation and auditing.	Depends on the application layer for inventory data and the operating system's default browser for displaying reports.

Data Decomposition

The data within WinChangeMonitor is structured to reflect the logical flow of operations from configuration through monitoring to reporting. Each class is designed to manage a specific layer of data abstraction, ensuring that user configurations, captured inventories, and computed differences remain clearly separated. This separation prevents duplication of information and promotes a clean data flow that mirrors the functional behavior of the system. By organizing data into hierarchical categories, the system maintains consistency between monitoring sessions and simplifies both serialization and change analysis.

At the highest level, configuration data defines the scope of monitoring, such as which files, folders, registry keys, or services are tracked. The next layer consists of inventory data, which represents the actual state of the system captured during pre- and post-execution scans. These inventories are compared to produce difference sets that highlight added, modified, or deleted items. The final layer represents reporting data, where the identified differences are aggregated, summarized, and formatted into human-readable reports. This decomposition ensures that transient runtime data remains distinct from persistent configuration data, thereby improving maintainability and reducing the potential for data corruption. This layered data structure simplifies the logic required for scanning and reporting and aligns with the project's modular design, making future enhancements or additional monitoring types easy to integrate without refactoring core logic.

Data Category	**Description**	**Primary Classes**
Configuration Data	Defines user-selected monitoring scopes, including directories, registry keys, and service lists.	FileSystemSettings, RegistrySettings, ServicesSettings
Inventory Data	Captures system state snapshots during baseline and post-installation scans.	FileSystemEntryInfo, RegistryEntryInfo, ServiceInfo
Difference Data	Represents computed changes between baseline and post-execution inventories, categorized as new, modified, or deleted.	RetainedSettings (Derived objects)
Reporting Data	Aggregates results for visualization and report generation, providing summaries for user review.	RetainedSettings, CommonInfo

Dependency Relationships

The dependency structure in WinChangeMonitor has been intentionally designed to minimize coupling and ensure clear lines of responsibility between modules. The system follows a top-down relationship model, where higher-level components manage workflow coordination and lower-level components handle specialized data operations. Dependencies are mostly unidirectional, allowing each component to operate independently within its defined scope while maintaining compatibility with the larger architecture.

The RetainedSettings class serves as the central controller and primary point of coordination, maintaining references to the three settings aggregates that handle file system, registry, and service monitoring. Each settings aggregate manages its own domain-specific objects, including tracked entities and corresponding inventories. The user interface depends on RetainedSettings and CommonInfo for displaying results and progress but does not directly interact with the domain or infrastructure layers. Serialization is managed independently through MessagePack utilities, which depend on annotated attributes within the data model but not on application logic. This structure ensures flexibility and reduces the risk of circular dependencies between modules.

Concurrency Description

The main goal of WinChangeMonitor's concurrency is to keep the user interface responsive while carrying out possibly time-consuming inventory scans. The design makes sure that file system and registry traversal, which can involve thousands of entries, is done on background threads so that the main interface thread can be used for status updates and user interaction. This method keeps the application from freezing or becoming unresponsive during intensive scanning operations while still offering a seamless user experience.

The concurrency model follows a single-producer, single-consumer pattern, where scanning threads produce inventory data and the user interface consumes status updates and progress indicators. Cross-thread communication is handled safely using asynchronous delegates that marshal calls back to the main thread through Invoke or BeginInvoke. Shared data structures such as the current inventory lists are protected by synchronized access when being written to, while read operations use cached snapshots to prevent blocking. Once a scan is complete, subsequent operations such as difference computation and report generation occur sequentially, eliminating the need for complex synchronization. This concurrency strategy ensures predictable and stable performance across various system configurations.

Concurrent Component	**Process Description**	**Concurrent Handling Mechanism**
Inventory Scanning	Runs background tasks to capture system states during pre- and post-execution phases.	Executes on background threads separate from the WinForms UI thread.
UI Updates	Reflects real-time progress and scan completion status.	Uses Invoke or BeginInvoke to safely update UI controls.
Data Access	Handles writing and reading of inventory data during scanning.	Uses locks or immutable snapshots to maintain thread safety.
Serialization & Reporting	Saves data and generates reports after scanning completes.	Executes sequentially after background threads finish, avoiding conflicts.

Design Patterns
WinChangeMonitor's architecture makes use of a number of proven software design patterns to enhance scalability, maintainability, and implementation clarity. The RetainedSettings class, which serves as a single point of access for configuration, inventory, and reporting functions, implements the Facade pattern. By offering a single API, this streamlines communication between the user interface and the underlying modules. In the user interface layer, where status indicators and progress updates react to background scanning events, the Observer pattern is used implicitly. The application logger employs the Singleton pattern to guarantee that audit and diagnostic messages are managed by a single logging instance during execution. In addition, the Data Transfer Object (DTO) pattern is reflected in classes such as FileSystemEntryInfo and RegistryEntryInfo, which encapsulate structured data that moves between modules without containing business logic. Together, these patterns promote consistency and reduce coupling between components, aligning with best practices for desktop software architecture.

NOTE: See ***figure 6*** in the appendix for the class diagram of the WinChangeMonitor

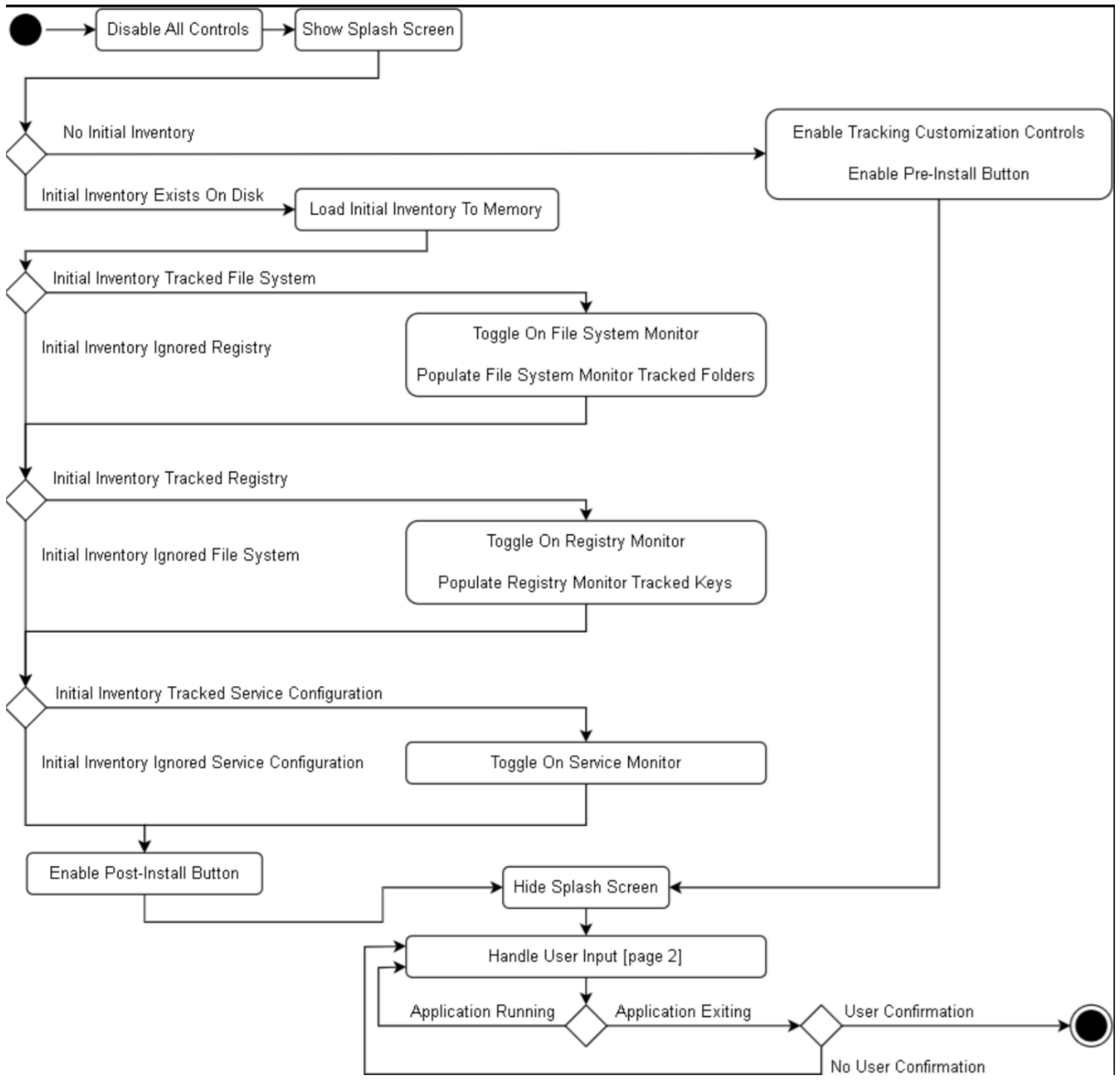
PERFORMANCE CONSIDERATIONS
The system is built to function well even when handling big inventories. In order to accomplish this, directory enumeration will be done using a breadth first strategy to guarantee that the user interface stays responsive throughout scans, and input and output operations will be carried out in batches. The program can choose to use a SQL database rather than just JSON or XML files when working with very large datasets, which usually have more than a million entries. In internal testing, switching from LightningDB to MessagePack serialization reduced the pre-install inventory time from roughly 1 hour 30 minutes to about 6 minutes 20 seconds, and replacing JSON deserialization with MessagePack deserialization reduced persistent data load time from about 7 minutes 17 seconds to under 17 seconds. This will save memory and speed up processing. Debouncing techniques will optimize interface updates during scanning, guaranteeing that status information and progress indicators are presented effectively without taxing system resources.

DESIGN CONSTRAINTS AND OUT-OF-SCOPE ITEMS
The project will not expand into advanced forensic analysis, integration with enterprise-scale SIEM solutions, real-time malware detection and remediation, or platform support beyond the Windows family of operating systems. The software will only run on Windows operating systems and will not support Linux or macOS platforms. These areas are considered out of scope in order to maintain focus on the project's objectives. By defining these constraints and boundaries, the project ensures that WinChangeMonitor will deliver a reliable and usable tool that supports developers, system administrators, and security analysts in managing risks associated with third-party executables, while staying aligned with the available time, resources, and technical constraints.

FUTURE EVOLUTION
Future updates to the application may include expanding the available export formats and enhancing the user interface with more advanced filtering and search capabilities. These improvements have already been identified as optional or "wish" items as seen in the requirements list for potential implementation in later development phases.

APPENDIX

State Diagrams



Disable All Controls
Show Splash Screen
No Initial Inventory
Enable Tracking Customization Controls
Enable Pre-Install Button
Initial Inventory Exists On Disk
Load Initial Inventory To Memory
Initial Inventory Tracked File System
Initial Inventory Ignored Registry
Toggle On File System Monitor
Populate File System Monitor Tracked Folders
Initial Inventory Tracked Registry
Initial Inventory Ignored File System
Toggle On Registry Monitor
Populate Registry Monitor Tracked Keys
Initial Inventory Tracked Service Configuration
Initial Inventory Ignored Service Configuration
Toggle On Service Monitor
Enable Post-Install Button
Hide Splash Screen
Handle User Input [page 2]
Application Running
Application Exiting
User Confirmation
No User Confirmation

Figure 3: Application Initialization State Diagram

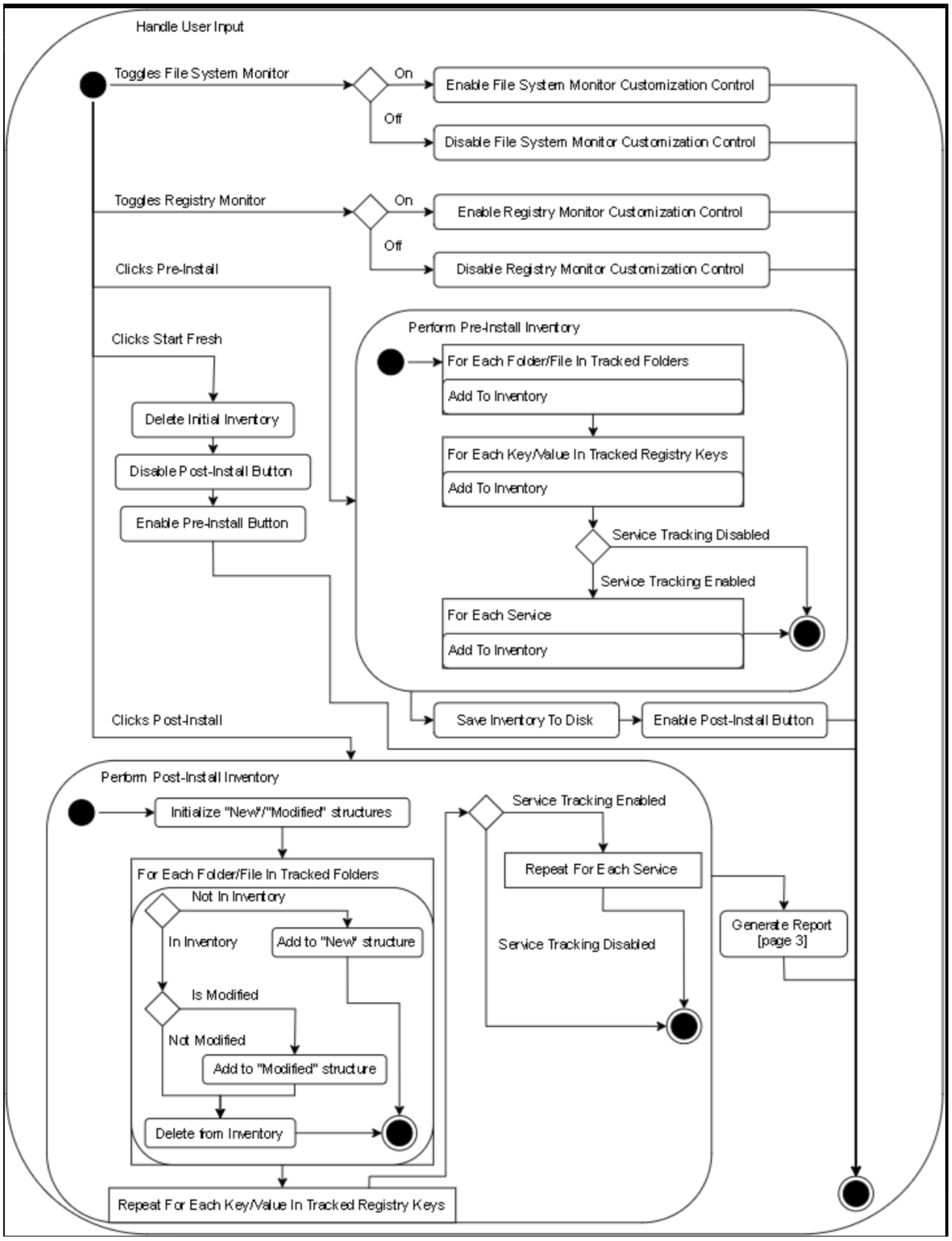
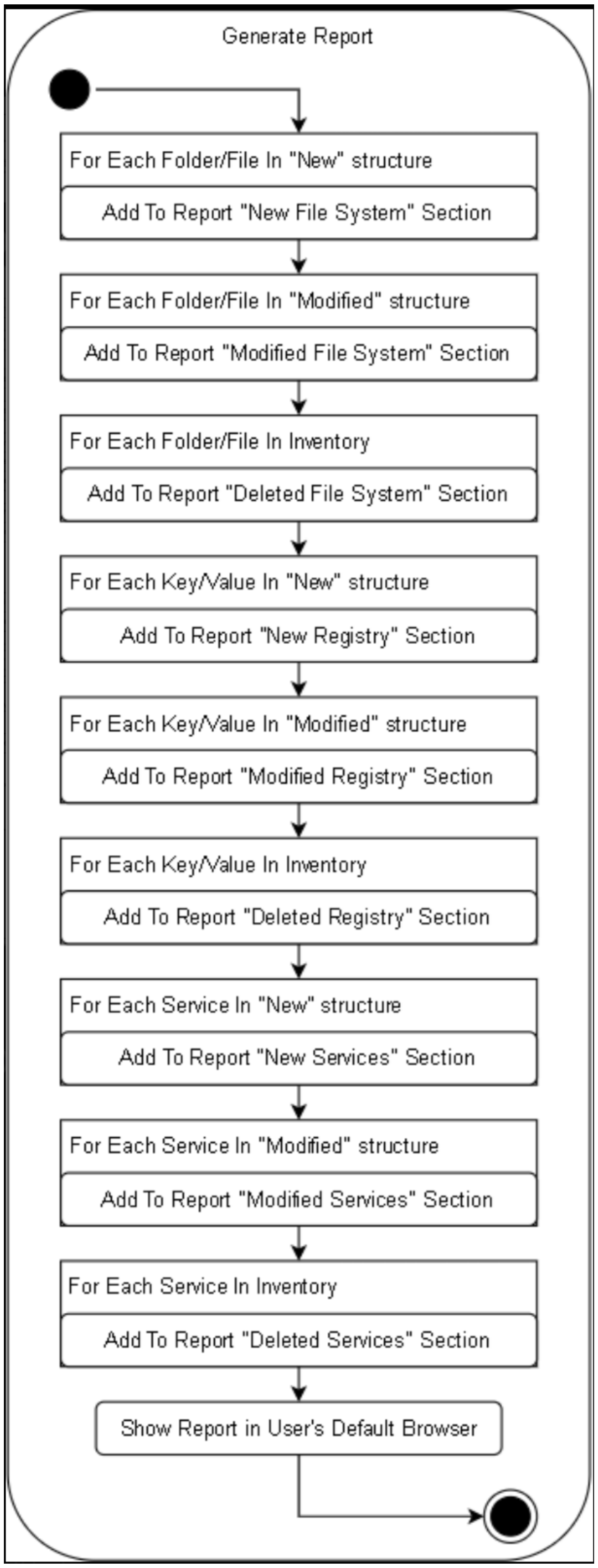

Handle User Input
Toggles File System Monitor
On
Off
Enable File System Monitor Customization Control
Disable File System Monitor Customization Control
Toggles Registry Monitor
On
Off
Enable Registry Monitor Customization Control
Disable Registry Monitor Customization Control
Clicks Pre-Install
Clicks Start Fresh
Delete Initial Inventory
Disable Post-Install Button
Enable Pre-Install Button
Perform Pre-Install Inventory
For Each Folder/File In Tracked Folders
Add To Inventory
For Each Key/Value In Tracked Registry Keys
Add To Inventory
Service Tracking Disabled
Service Tracking Enabled
For Each Service
Add To Inventory
Save Inventory To Disk
Enable Post-Install Button
Clicks Post-Install
Perform Post-Install Inventory
Initialize "New"/"Modified" structures
For Each Folder/File In Tracked Folders
Not In Inventory
In Inventory
Add to "New" structure
Is Modified
Not Modified
Add to "Modified" structure
Delete from Inventory
Repeat For Each Key/Value In Tracked Registry Keys
Service Tracking Enabled
Repeat For Each Service
Service Tracking Disabled
Generate Report [page 3]

Figure 4: User Interaction and Inventory Management State Diagram


Generate Report
For Each Folder/File In "New" structure
Add To Report "New File System" Section
For Each Folder/File In "Modified" structure
Add To Report "Modified File System" Section
For Each Folder/File In Inventory
Add To Report "Deleted File System" Section
For Each Key/Value In "New" structure
Add To Report "New Registry" Section
For Each Key/Value In "Modified" structure
Add To Report "Modified Registry" Section
For Each Key/Value In Inventory
Add To Report "Deleted Registry" Section
For Each Service In "New" structure
Add To Report "New Services" Section
For Each Service In "Modified" structure
Add To Report "Modified Services" Section
For Each Service In Inventory
Add To Report "Deleted Services" Section
Show Report in User's Default Browser
Figure 5: Report Generation State Diagram

Class Diagram

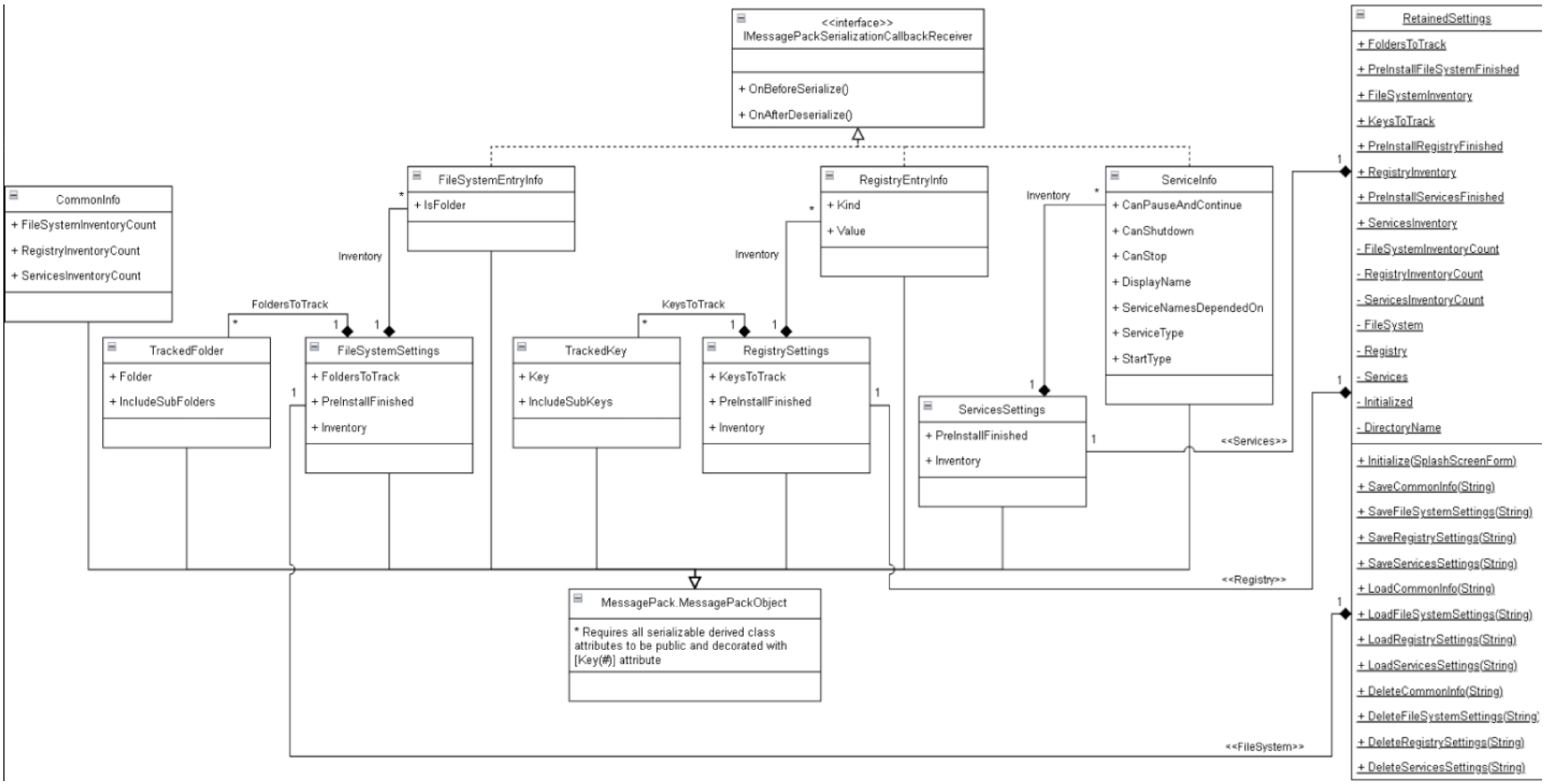
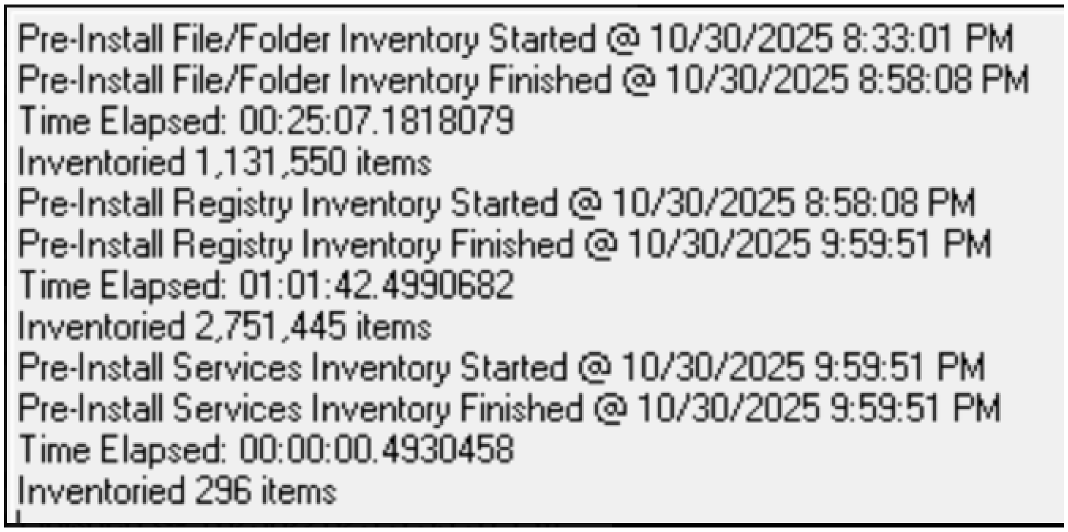


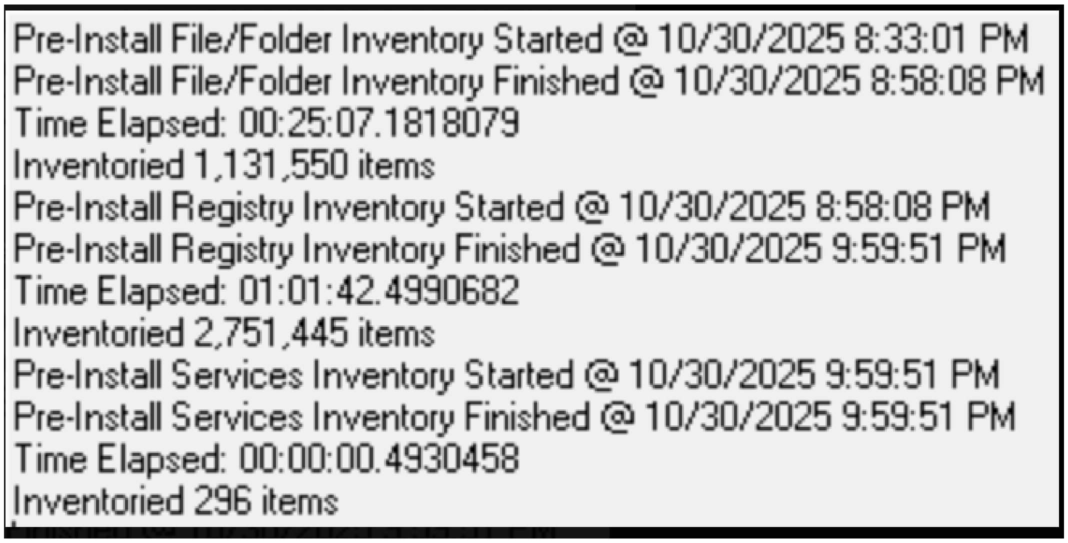
Figure 6: Class Diagram for the WinChangeMonitor

Benchmark Results



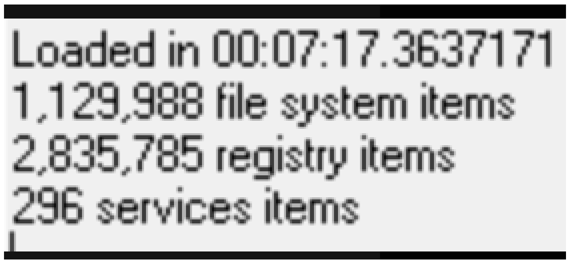
```
Pre-Install File/Folder Inventory Started @ 10/30/2025 8:33:01 PM
Pre-Install File/Folder Inventory Finished @ 10/30/2025 8:58:08 PM
Time Elapsed: 00:25:07.1818079
Inventoried 1,131,550 items
Pre-Install Registry Inventory Started @ 10/30/2025 8:58:08 PM
Pre-Install Registry Inventory Finished @ 10/30/2025 9:59:51 PM
Time Elapsed: 01:01:42.4990682
Inventoried 2,751,445 items
Pre-Install Services Inventory Started @ 10/30/2025 9:59:51 PM
Pre-Install Services Inventory Finished @ 10/30/2025 9:59:51 PM
Time Elapsed: 00:00:00.4930458
Inventoried 296 items
```

Figure 7: LightningDB benchmark



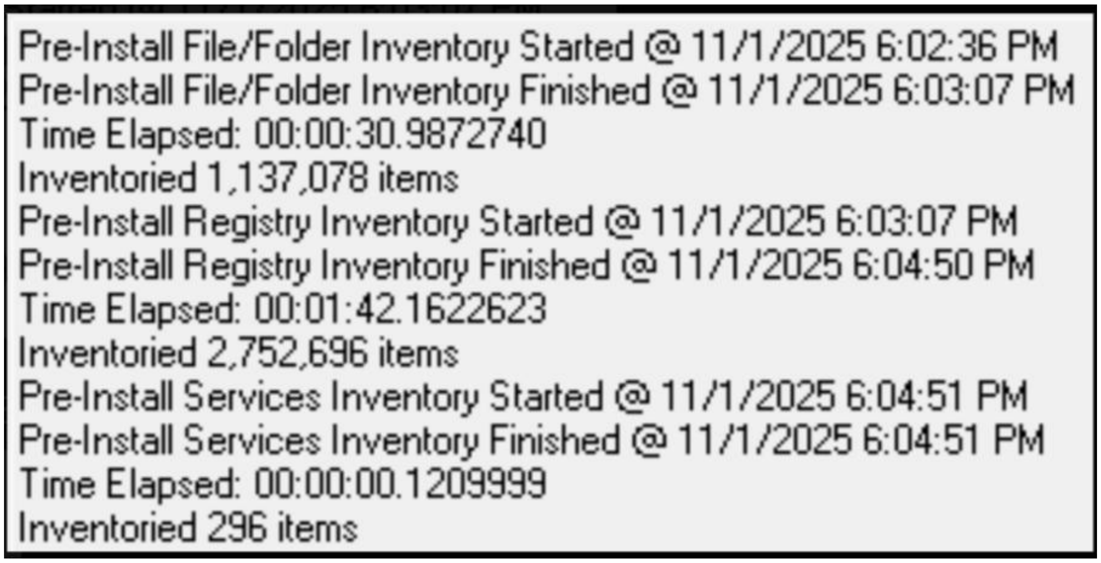
```
Pre-Install File/Folder Inventory Started @ 10/30/2025 8:33:01 PM
Pre-Install File/Folder Inventory Finished @ 10/30/2025 8:58:08 PM
Time Elapsed: 00:25:07.1818079
Inventoried 1,131,550 items
Pre-Install Registry Inventory Started @ 10/30/2025 8:58:08 PM
Pre-Install Registry Inventory Finished @ 10/30/2025 9:59:51 PM
Time Elapsed: 01:01:42.4990682
Inventoried 2,751,445 items
Pre-Install Services Inventory Started @ 10/30/2025 9:59:51 PM
Pre-Install Services Inventory Finished @ 10/30/2025 9:59:51 PM
Time Elapsed: 00:00:00.4930458
Inventoried 296 items
```

Figure 8: JSON serialization benchmark



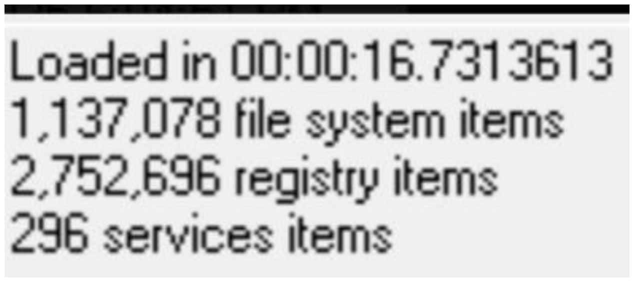
```
Loaded in 00:07:17.3637171
1,129,988 file system items
2,835,785 registry items
296 services items
```

Figure 9: JSON deserialization benchmark



```
Pre-Install File/Folder Inventory Started @ 11/1/2025 6:02:36 PM
Pre-Install File/Folder Inventory Finished @ 11/1/2025 6:03:07 PM
Time Elapsed: 00:00:30.9872740
Inventoried 1,137,078 items
Pre-Install Registry Inventory Started @ 11/1/2025 6:03:07 PM
Pre-Install Registry Inventory Finished @ 11/1/2025 6:04:50 PM
Time Elapsed: 00:01:42.1622623
Inventoried 2,752,696 items
Pre-Install Services Inventory Started @ 11/1/2025 6:04:51 PM
Pre-Install Services Inventory Finished @ 11/1/2025 6:04:51 PM
Time Elapsed: 00:00:00.1209999
Inventoried 296 items
```

Figure 10: MessagePack serializaton benchmark



```
Loaded in 00:00:16.7313613
1,137,078 file system items
2,752,696 registry items
296 services items
```

Figure 11: MessagePack deserialization benchmark

Team Contributions

Jeff Rose: I created the State diagram and the Class diagram, in addition to reviewing and providing feedback for the SDD. For the Github repo, I benchmarked several different data persistence technologies and optimized a solution that reduces the Pre-Install Inventory time from roughly 1 hour, 30 minutes with LightningDB to roughly 6 minutes, 20 seconds with MessagePack serialization and reduces the persistent data load time from disk from 7 minutes, 17 seconds with JSON deserialization to under 17 seconds with MessagePack deserialization.

Anjian Chen: The team met on November 2nd to reflect on our midterm presentation and discuss SDD document. I proposed we should include user instructions on the steps to use our application in the README.md file. I also pointed out that we did a great job on presentation timing and delivery but need improvements on retrospective evaluation. I record the notes of our meeting for SDD and future references. I also started testing on codes we have. Planning to write more unit tests once we learn about them more during lecture.

Yu Wu: I provided several suggestions for the SDD document, particularly offering some improvement recommendations for the UI section.

Yeryoung Kim: As my contribution, I reviewed the SDD and created a Layered Architecture Diagram to include in the appendix under the High-Level Design section, helping to improve clarity and visual understanding of the system's structure. I also suggested briefly mentioning any design patterns applied in the system—for example, noting that if the UI automatically updates in response to background scanning events, it likely follows the Observer pattern, which I believe our system might be using.

Princely Oseji: Collaborated with the team to create the project proposal, combined SCMP/SPMP, SRS and the SDD documents. Created the initial drafts for these documents based on discussions during the stand-up meetings and modified them according to feedback given by both the team and the professor until ready for submission.

GITHUB: https://github.com/pco30/Software_Eng