Concrete Architecture of GNUstep: A Detailed Analysis of libs-base

Alex Liang, Chaolin Zhao, Lucas Chu, Simone Jiang, Yueyi Huang, Zhongren Zhao
Queen's University
CISC 322/326 Software Architecture
Instructor*:* Bram Adams
TA: Mohammed Mehedi Hasan
Date*: 04/04/2025*

1. Abstract

This report presents a detailed analysis of the concrete architecture of the libs-base subsystem within GNUstep and proposes a telemetry-based enhancement aimed at improving performance diagnostics and user experience. The lack of built-in data collection tools in GNUstep limits developers' ability to efficiently detect and resolve system issues. To address this, we introduce an automatic telemetry mechanism that captures interaction patterns and performance metrics, enabling empirical, data-driven improvements. The architecture is designed to preserve modularity and maintain compatibility with GNUstep's object-oriented structure. We assess the impact of this enhancement on system qualities—such as maintainability, reliability, and performance—and explore architectural alternatives including object-oriented and layered models. Our evaluation demonstrates that the proposed telemetry system strengthens GNUstep's robustness and adaptability while raising important considerations regarding privacy, scalability, and long-term maintainability.

2. Introduction

2.1 Current State of GNUstep

GNUstep is a mature, open-source development environment rooted in the Cocoa framework, enabling cross-platform application development with an object-oriented approach. It provides a modular architecture consisting of key subsystems, with libs-base serving as the foundation for utilities such as memory management, notifications, and preferences. Despite its extensibility, GNUstep currently lacks built-in mechanisms for collecting real-time telemetry data—limiting developers' ability to proactively detect, diagnose, and address performance bottlenecks or user experience issues. This gap in observability hinders maintainability and responsiveness, especially in complex environments. To improve system insight and developer feedback loops, our team proposes a telemetry enhancement focused on modernizing the libs-base subsystem while preserving GNUstep's architectural principles.

2.2 Summary of the Report

This report introduces an architectural enhancement to GNUstep's **libs-base** subsystem through the integration of an automated telemetry system. Designed to capture user interactions and system performance metrics, the enhancement aims to support evidence-based debugging and informed optimization. We explore the proposed design in detail, focusing on its integration within existing modules and the introduction of a centralized telemetry manager. The report evaluates two implementation strategies—object-oriented and layered architectures—and assesses their trade-offs using SAAM analysis across non-functional requirements such as maintainability, security, performance, and testability. Finally, we examine the impact of the enhancement on both high-level and low-level architectural elements, concluding with considerations for long-term evolution, user privacy, and system reliability.

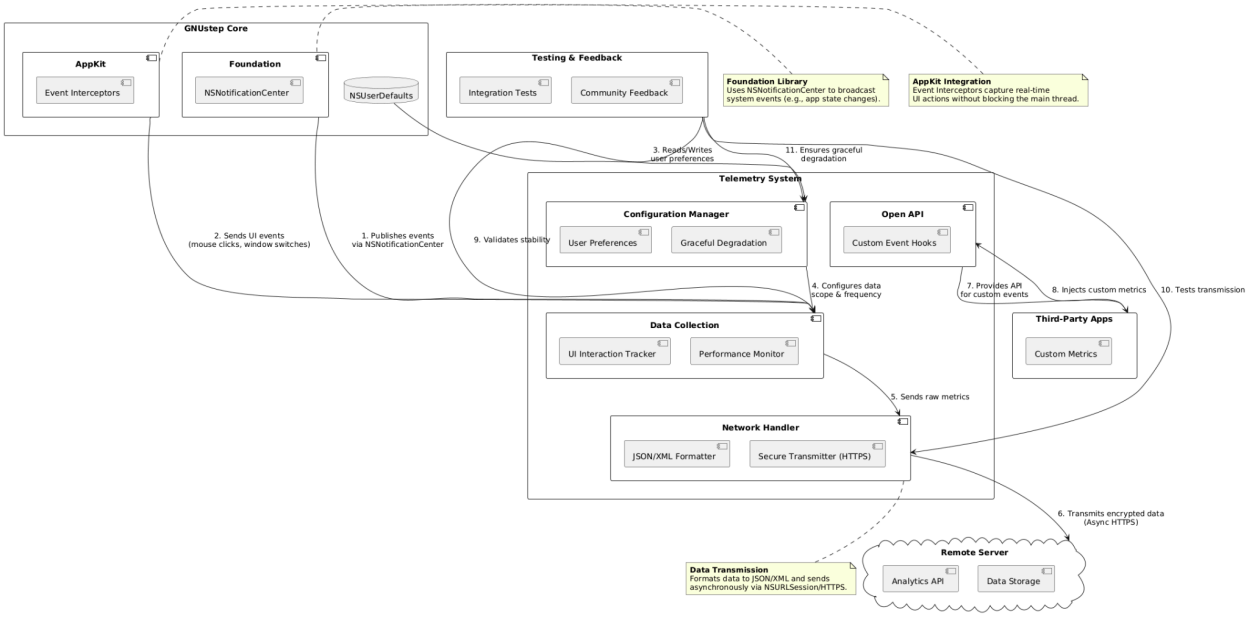
3. Proposed Enhancement

3.1 Motivation

The introduction of automatic telemetry in GNUstep aims to address critical gaps in diagnosing issues and optimizing performance by systematically collecting user interaction and system performance data. Currently, GNUstep lacks robust mechanisms to monitor real-time user behavior or system metrics, forcing developers to rely on fragmented manual reports or reactive debugging, which often lack the context or consistency needed to resolve complex issues. Automated telemetry resolves this by capturing structured data on interactions and performance metrics, enabling developers to pinpoint bottlenecks like inefficient event-handling logic or memory leaks triggered by specific actions.By transforming subjective feedback into empirical insights, telemetry not only accelerates issue resolution but also supports proactive, evidence-based development, ensuring updates avoid performance regressions and align with user needs.

By streaming performance metrics across different environments (e.g., OS-specific CPU spikes or memory leaks), developers can learn about problems immediately, allowing them to quickly fix issues before they impact users (e.g., deploy a patch for a Linux-specific resource bottleneck). Aggregated data can also inform strategic planning, revealing trends such as underutilized features or workflows that need optimization, guiding the prioritization of updates (e.g., improving lag-prone tools that users often abandon). For the community, anonymized usage data ensures that development is aligned with real needs (e.g., maintaining old features favored by long-time users or streamlining niche tools into plugins), fostering engagement and inclusion.

3.2 Implementation



To integrate telemetry effectively with GNUstep, the NSNotificationCenter component from the Foundation library is used to broadcast events from within the core system. This allows the telemetry subsystem to subscribe and respond to system-level . At the same time, the libs-gui is modified using method swizzling to embed event hooks at key UI interaction points so that user behavior is captured

in real time. Importantly, these interceptors are designed to operate on background threads or asynchronously, ensuring that UI responsiveness is preserved and UI jitter is avoided.

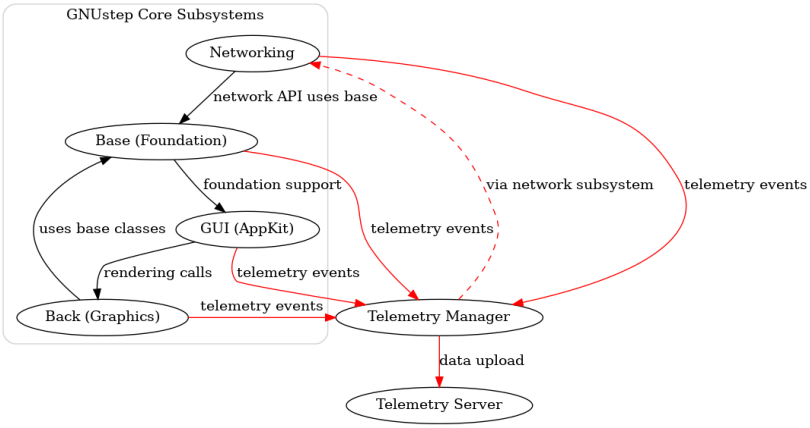
Telemetry settings are managed through the NSUserDefaults configuration mechanism, enabling users to control opt-in/opt-out status, data collection frequency, and the scope of collected metrics. This approach not only respects user consent and GDPR compliance but also makes telemetry adaptable to different usage scenarios. Furthermore, the configuration logic is designed with compatibility in mind, supporting multiple GNUstep versions and environments. This flexibility ensures that even if a platform lacks advanced features, the system can degrade gracefully without introducing instability or unexpected behavior.

The telemetry subsystem extends GNUstep's networking modules such as NSURLSession to transmit collected data securely and asynchronously using HTTPS/TLS 1.3. This ensures that all telemetry transmissions occur in the background, avoiding blocked threads or interruptions to application workflows. The data is serialized using standardized formats like JSON/XML, which allows for easy parsing, analysis, and future extensibility on remote servers or analytics platforms following a client-server architecture.

To support extensibility, the telemetry subsystem provides an Open API that allows third-party libraries and higher-level applications to register custom event hooks. These hooks enable developers to track specific domain or business-level metrics without modifying the telemetry core. In parallel, care is taken to prevent redundancy by coordinating with existing analytics or monitoring tools, ensuring a consistent modular architecture and avoiding duplicate metric capture. The use of a shared data format ensures compatibility and cohesion across all telemetry modules.

During development, telemetry modules undergo thorough unit testing and integration testing to verify component behavior and stability across scenarios. Stress testing is also conducted to simulate high user interaction loads and monitor any potential effects on performance, particularly for UI responsiveness. The telemetry subsystem adopts strategies such as exponential backoff for failed transmissions and write-ahead logging (WAL) in SQLite to ensure reliable and resilient data buffering. Community feedback is regularly incorporated to improve telemetry design, aligned with GNUstep's evolving ecosystem and long-term maintainability.

4. The Current State Of System



The current conceptual architecture for GNUstep retains the same modular, layered, and object-oriented structure defined before. The system continues to follow a hybrid architecture composed of key subsystems, including libs-base, libs-gui, libs-back, apps-gorm. Each layer remains responsible for distinct concerns: libs-base provides foundational services, libs-gui manages user interface components, libs-back handles rendering, and Networking delivers core communication functionality. Our proposed telemetry enhancement will not introduce a new subsystem; instead, it will be implemented as an additive feature across existing subsystems, primarily through new event emitters and a centralized **Telemetry Manager** integrated within the libs-base subsystem.

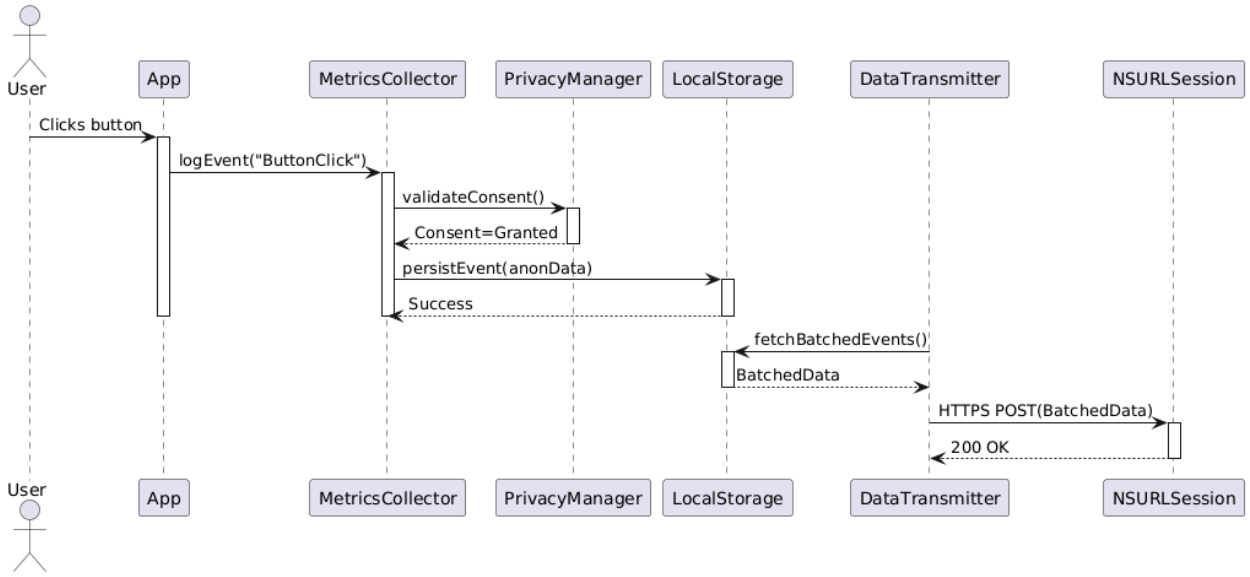
While no structural subsystems are changed in the conceptual architecture, this enhancement introduces one new **cross-cutting dependency**, shown in red in the figure below. Specifically, we introduce a libs-base → Remote Telemetry Server dependency to enable secure, asynchronous data transmission. This addition allows the system to collect runtime metrics—such as user interaction events from libs-gui, rendering performance data from libs-back, and network usage patterns from Networking, and transmit them for external analysis without altering the core logic of each subsystem.

Our enhancement leverages the existing **NSNotificationCenter** in libs-base *(GNUstep MediaWiki, n.d.)* to register and dispatch telemetry events across modules, allowing a non-intrusive integration that preserves maintainability. It also uses NSUserDefaults to expose configuration options, such as telemetry opt-in and data frequency, ensuring user control and regulatory compliance. These integrations occur within the boundaries of each subsystem, and the telemetry feature behaves as a passive observer that collects and sends information in parallel to the system's main workflow.

In summary, GNUstep's conceptual architecture remains unchanged at the subsystem level, but is **extended with a new dependency** that enables remote telemetry. The integration aligns with GNUstep's existing object-oriented principles and layered communication patterns, ensuring that telemetry can be evolved or removed without affecting the structural integrity of the system.

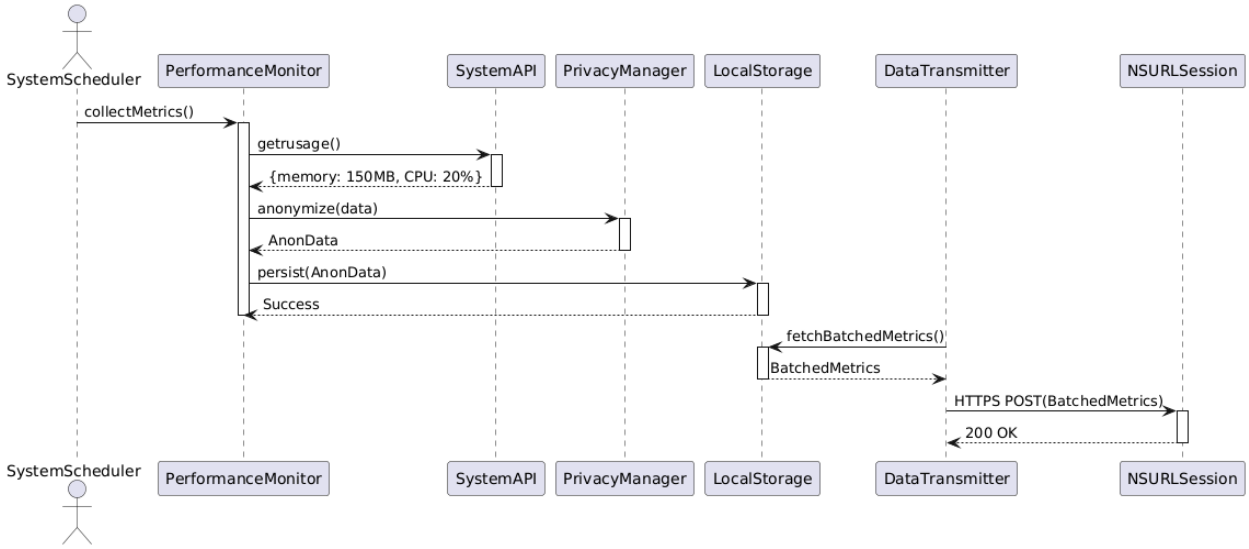
5. Use Cases

5.1. Sequence Diagram for User Interaction Tracking



This diagram illustrates the workflow for capturing and transmitting user interaction events (e.g., a button click) in the telemetry system. The process begins when a user triggers an action (e.g., clicking a button), prompting the application to call the logEvent("ButtonClick") method on the MetricsCollector. The MetricsCollector first verifies user consent by querying the PrivacyManager, which confirms whether data collection is allowed. If consent is obtained, the MetricsCollector anonymizes the event data (e.g., strips PII) and saves it to LocalStorage (e.g., SQLite), ensuring that the data survives an application crash or restart. The DataTransmitter asynchronously and periodically fetches batches of events from LocalStorage. Once retrieved, it initiates an encrypted HTTPS POST request through NSURLSession to transmit the data to a remote server. Using background threads for storage and transmission ensures that the main UI thread remains free, thereby maintaining the responsiveness of the application.

5.2. Sequence Diagram for Performance Monitoring



This diagram outlines the workflow for collecting and transmitting system performance metrics (e.g., memory/CPU usage) in the telemetry system. The process begins when the SystemScheduler (e.g., a timer) triggers the PerformanceMonitor to collectMetrics(). The PerformanceMonitor queries the SystemAPI (e.g., via getrusage()) to gather raw resource statistics, such as memory consumption (150MB) and CPU utilization (20%). This data is then forwarded to the PrivacyManager, which anonymizes it—for example, hashing device identifiers or stripping contextual details for security.

The anonymized metrics are persisted to LocalStorage (e.g., SQLite), ensuring durability across app restarts or crashes. Asynchronously, the DataTransmitter retrieves batched metrics from LocalStorage and initiates an encrypted HTTPS POST request via NSURLSession to transmit the data to a remote server. The server acknowledges successful receipt with a 200 OK response.

6. The Effects on System Qualities

6.1. Maintainability

The telemetry subsystem enhances maintainability through its modular architecture, decoupling components like MetricsCollector, PrivacyManager, and DataTransmitter to enable independent updates via well-defined interfaces (e.g., modifying anonymization logic without altering network module code). Centralized configuration via Info.plist simplifies settings management, reducing hardcoded values and streamlining environment-specific adjustments (e.g., staging vs. production). However, method swizzling in foundational classes (e.g., NSApplication) introduces fragility, as future GNUstep updates may break swizzled event handlers, necessitating urgent fixes. Additionally, tight coupling to SQLite for persistence complicates schema migrations, requiring backward-compatible strategies to avoid data corruption. While modularity improves scalability and flexibility, these risks demand rigorous testing to balance adaptability with long-term system resilience.

6.2. Evolvability

The telemetry subsystem’s extensible API and modular design enable developers to introduce custom metrics (e.g., logging domain-specific actions like logCustomEvent("DocumentSaved")) without modifying core logic. Third-party libraries can leverage this API while preserving the integrity of the telemetry infrastructure. The decoupled network module (e.g., DataTransmitter) isolates transmission logic, allowing protocol changes (e.g., HTTPS to MQTT) with minimal impact on data collection or storage. This separation adheres to the open/closed principle, supporting iterative evolution while maintaining stability in existing features.

6.3. Testability

Testability is enhanced by the modular architecture and debug features (e.g., local SQLite storage for offline validation). The NSTelemetryManager centralized data handling, enabling isolated unit tests for buffering, batching, and retry mechanisms. A compile-time flag (GS_ENABLE_TELEMETRY) disables telemetry during testing to eliminate side effects. Dependency injection mocks asynchronous operations (e.g., simulating network failures via stubs). However, validating event capture hooks in GUI components (e.g., NSApplication.m) requires integration testing, as unit tests may miss edge cases from GNUstep’s internal event propagation or thread

synchronization. UI automation tests replicate real user behavior to validate swizzled methods in classes like NSControl, though these tests are slower and harder to maintain.

6.4. Performance

The telemetry subsystem optimizes performance by delegating resource-intensive tasks (e.g., SQLite writes, HTTPS transfers via NSURLSession) to background threads, preventing main-thread blocking and preserving UI responsiveness. Batching events minimizes network overhead, reduces latency, and conserves battery life. However, method swizzling in GUI components adds marginal runtime overhead, potentially degrading responsiveness in apps with dynamic UIs . Frequent SQLite writes may strain devices with slow storage, causing jitter during write-heavy operations. While write-ahead logging (WAL) mitigates I/O contention, low-end devices may still experience latency spikes, necessitating platform-specific optimizations in build scripts (e.g., GNUmakefile).

6.5. Security

The subsystem enforces security via anonymization (e.g., hashing device IDs) before buffering data in SQLite, reducing exposure risks even if storage is compromised. However, unencrypted SQLite files pose local storage risks—attackers with physical/jailbroken device access could tamper with or extract metrics. Event hooks in libs-gui risk inadvertent PII capture (e.g., clipboard data) if inputs are not sanitized pre-anonymization. The network layer uses HTTPS/TLS 1.3 as a more secure and rapid method to transmission(*IBM, 2025, February 14.*), but misconfigured server endpoints in GNUstepDefaults.h could become attack vectors. Compile-time flags disable telemetry for high-security deployments, while logging safeguards prevent accidental data leaks. Future mitigations include SQLite encryption and input sanitization in GUI handlers.

6.6. Reliability

Reliability is ensured via an offline queue that atomically writes metrics to SQLite, preventing data loss during crashes or network outages. Retry logic recovers from transient failures, ensuring eventual delivery. However, unbounded SQLite storage risks disk space exhaustion, crippling low-storage devices. Retention policies and compile-time flags for ephemeral telemetry mitigate this risk. Write-ahead logging (WAL) is enabled by default to prevent I/O contention, while atomic event logging in UI hooks avoids partial data loss. Proactive configuration safeguards are critical to balance reliability with resource constraints.

7. Architectural Alternatives

This section evaluates the Client-Server and Layered architectures as viable alternatives, analyzing their trade-offs in scalability, security, and system integration. By contrasting these models against the chosen Object-Oriented approach, we illuminate their suitability for specific use cases, such as large-scale data aggregation or regulated environments, while underscoring the importance of balancing flexibility with GNUstep’s design ethos.

7.1 Client-Server Architecture

The Client-Server Architecture divides the telemetry system into two distinct roles: the client, embedded within GNUstep applications, and a centralized server for data processing. The client, managed by the NSTelemetryManager, collects user interactions and performance metrics, then transmits them asynchronously via NSURLSession or NSURLConnection over HTTPS. The server handles authentication, storage, and analytics, often using REST APIs or databases. This model centralizes data management, simplifying compliance and large-scale analysis, while enabling independent scaling of server resources. However, it introduces network dependency—telemetry fails without internet connectivity—and a single point of failure if the server goes offline. For GNUstep, this approach suits enterprise environments requiring centralized oversight but conflicts with its decentralized design philosophy, as it relies heavily on external infrastructure.

7.2 Layered Architecture

The Layered Architecture introduces a dedicated Telemetry Layer atop GNUstep's core layers (Foundation, AppKit), isolating telemetry logic from underlying components. This layer interacts with **NSNotificationCenter** for event capture and NSUserDefaults for configuration, communicating via standardized APIs like NSURLSession. While this enforces separation of concerns and simplifies maintenance, inter-layer communication introduces latency, and rigid boundaries may hinder rapid iteration. For example, adding new telemetry features might require cross-layer adjustments, complicating development. This model suits environments requiring strict modularity (e.g., regulatory compliance) but struggles with flexibility, making it less ideal for dynamic or evolving use cases within GNUstep's ecosystem.

8. SAAM Analysis

8.1 Stakeholders Analysis

Given the scale of the GNUstep project, its open-source nature, and the scope of telemetry enhancement, the architecture must meet the expectations of a diverse set of stakeholders:

- **Core Developers**: These contributors focus on code quality, maintainability, and alignment with GNUstep's modular philosophy. Their primary concern is whether the telemetry system integrates cleanly into existing components without introducing fragility or long-term technical debt.
- **End Users**: As the primary source of interaction data, users demand transparency, control, and performance. Their trust is built through clear opt-in mechanisms, data privacy guarantees (e.g., GDPR compliance), and minimal performance impact during usage.
- **Third-party Developers and Open-Source Contributors**: These stakeholders look for extensible APIs and flexible integration points. A telemetry solution must be modular and sufficiently documented to support community-driven extensions.
- **System Maintainers**: Responsible for long-term upkeep, they prioritize stability, backward compatibility, and low configuration overhead. They require a telemetry system that remains robust across updates and heterogeneous environments.
- **Privacy and Legal Compliance Teams**: These stakeholders enforce strict adherence to regional and international data privacy regulations. The architecture must support anonymization, encryption, and consent tracking to avoid compliance violations.

8.2 Non-Functional Requirements

The proposed telemetry enhancement introduces critical non-functional requirements (NFRs), which shape architectural choices: **Performance, Maintainability, Reliability, Security and Privacy, Scalability, and Testability.** (mentioned in 6.1 - 6.6)

8.3 Effects of Enhancement

Firstly, users and developers who value better visibility into application performance and user behavior will definitely benefit from the telemetry enhancement. If implemented thoughtfully, it could help identify bugs more efficiently, improve UI responsiveness, and guide future feature development based on real usage data. There would also likely be more engagement from open-source contributors, as the modular object-oriented design improves evolvability and testability. However, this enhancement introduces additional processing and storage demands, such as background data queuing and transmission, which may slightly impact performance on older devices. It also brings a certain level of maintenance overhead, as components like event hooks and data transmitters will require frequent testing and updates, especially when core GNUstep classes evolve. Moreover, since telemetry involves user data, strict adherence to privacy regulations like GDPR is essential, which means more design considerations around consent and anonymization.

8.4 Evaluating the Two Proposed Implementations

Two architectural strategies are considered for telemetry integration:

Implementation 1: Client-Server Architecture

This implementation offloads telemetry storage and analysis to a centralized server.

- **Strengths**:
  - Simplifies compliance and auditing through centralized data control.
  - Scalable for large user bases and enables real-time analytics.
  - Compatible with asynchronous upload strategies using NSURLSession.
- **Weaknesses**:
  - Dependent on continuous network availability.
  - Introduces a single point of failure (server downtime interrupts data flow).

Implementation 2: Layered Architecture

Telemetry is implemented as a self-contained layer interacting with GNUstep components via abstract interfaces.

- **Strengths**:
  - Maintains separation of concerns, reducing integration complexity.
  - Facilitates testability and debugging due to clear module boundaries.
  - Well-aligned with existing GNUstep object-based design.
- **Weaknesses**:

- Introducing cross-cutting telemetry features (e.g., global error tracking) becomes harder due to strict layer separation.
- Adds communication latency between layers.

NFR	Client-Server	Layered
Proformance	High	Moderate
Maintainability	Moderate	High
Reliability	High	Moderate
Privacy	High	High
Testability	Moderate	High
Scalability	Moderate	Moderate

Result: The Client-Server model provides stronger support for real-time monitoring, compliance, and scaling, while the Layered model favors long-term maintainability and isolation.

8.5 Conclusion on Emphases and Focuses

In conclusion, if we care more about scalability, centralized data handling, and making it easier to analyze user behavior over time, then the client-server approach makes the most sense. It gives us a clean way to manage data securely, stay compliant with privacy laws, and roll out improvements without needing to update every client. This comes with some trade-offs, like relying on a stable network and needing to maintain a server, but it's a solid and realistic choice for a project like this. If, instead, our top priority was keeping things simple and fully self-contained, a layered design is more worth exploring as an alternative solution.

9. Impact on Architectural Levels: High-Level & Low-Level

The integration of telemetry functionality into GNUstep significantly impacts both the high-level and low-level conceptual architectures. These effects are primarily observed through the introduction of new modules, modification of existing components, and enhancement of communication mechanisms.

9.1 High-Level Conceptual Architecture Effects

The telemetry subsystem introduces a structured metrics framework within GNUstep, anchored by the NSTelemetryManager class in the libs-base component. This manager coordinates data buffering, anonymization, and scheduling while integrating with libs-base (via NSNotificationCenter for system-wide events) and libs-gui (via UI event hooks in AppKit) to capture user interactions and system activities with minimal workflow disruption.

Network module enhancements leverage GNUstep's NSURLSession for secure, asynchronous data transmission, using standardized formats (e.g., JSON/XML) for cross-platform

compatibility and future extensibility. A modular architecture allows third-party extensions via public APIs (e.g., custom event tracking like logCustomEvent("DocumentSaved") without core logic changes, isolating protocol updates (e.g., HTTPS to MQTT) to preserve stability.

Configuration is streamlined via NSUserDefaults (user-controlled toggles, frequency adjustments) and compile-time flags (e.g., GS_ENABLE_TELEMETRY). These layers balance flexibility with GDPR compliance, ensuring alignment with GNUstep's modularity and efficiency principles while adapting to developer and regulatory needs.

9.2 Low-Level Conceptual Architecture Effects

The telemetry enhancement modifies core GNUstep files (e.g., NSApplication.m, NSEvent.m) with event hooks to intercept UI interactions (clicks, inputs) and system events, forwarding raw data to the NSTelemetryManager for anonymization and buffering. Network modules like NSURLSession are extended to handle encrypted, asynchronous HTTPS transmissions. These changes embed telemetry deeply into GNUstep's runtime, requiring careful synchronization to avoid conflicts with existing event propagation or thread logic.

Error handling is bolstered via retries (exponential backoff), SQLite's atomic writes for crash resilience, and logging for operational transparency. Graceful degradation ensures telemetry hooks bypass data collection when disabled via NSUserDefaults, minimizing overhead. Build configurations (GNUmakefile, GNUstepDefaults.h) enable compile-time exclusion of telemetry for lightweight deployments, while runtime settings enforce GDPR-compliant opt-in/opt-out. Privacy safeguards—stripping PII pre-storage and discarding raw input in event handlers—ensure secure, adaptable operation across environments.

These adjustments balance real-time data capture with GNUstep's architectural integrity, prioritizing reliability, compliance, and minimal intrusion.

10. Codebase Impact

10.1 Lib-base

New Telemetry Module Directory and Files
New Directory:
Create a dedicated subdirectory within the GNUstep Base component. This directory will house all source code related to telemetry functionality.
New Source Files:
NSTelemetryManager.h / NSTelemetryManager.m
These files manage data collection, buffer the telemetry data, and schedule its transmission to the remote server. They act as the core of the telemetry module.

Modifications to Existing Files
Performance Monitoring Integration:

If performance metrics need to be captured more granularly, modify existing performance monitoring components (such as NSProcessInfo or related modules) to include telemetry data collection hooks.
Configuration and Build Scripts:
Configuration Files (e.g., config.h or GNUstepDefaults.h):
Introduce a macro or flag to toggle telemetry functionality, allowing it to be enabled or disabled at compile time.
Build Scripts (e.g., GNUmakefile):
Update the build rules to include the new Telemetry directory and ensure that the new files are compiled and linked correctly within the GNUstep Base component.

10.2 Lib-GUI

Event Handling Enhancements
Main Application Event Loop: NSApplication.m, Insert hooks into the main event loop to capture global user interaction events. These hooks will forward event data to the telemetry manager for processing.
Specific Event Handlers: NSEvent.m or NSResponder.m, Enhance these files to embed data collection logic during the processing of specific user events (such as mouse clicks or keyboard inputs). This ensures that detailed telemetry data is captured whenever a user interacts with the application.

Control-Specific Adjustments (If Necessary)
Consider adding telemetry hooks within the individual implementation files of UI components like buttons or menu items. This provides fine-grained control over event tracking for specific elements.

10.3 Networking

Network Module Enhancements or New Module Creation
Existing Network Components:
If GNUstep includes network communication classes (e.g., NSURLConnection), extend these components to support uploading telemetry data.

New Network Module:
If no suitable network component exists, create a lightweight network module dedicated to transmitting telemetry data to the remote server.

Logging and Error Handling
Incorporate logging and error-handling mechanisms within the telemetry and network modules. This ensures that if there is a network failure or an error during data transmission, the system can gracefully handle the issue without negatively impacting the user experience.

11. Testing

To ensure seamless integration of the telemetry subsystem with GNUstep's existing features, a multi-layered testing strategy validates compatibility, data integrity, and performance. Unit testing targets core components like the NSTelemetryManager, verifying data collection, buffering, and anonymization logic. Tests simulate edge cases (e.g., network outages, malformed events) to confirm error recovery. Event hooks in modified libs-gui files (e.g., NSApplication.m, NSEvent.m) are rigorously validated to ensure UI interactions trigger telemetry events without disrupting GNUstep's native event propagation. Integration testing evaluates interactions between telemetry and critical subsystems: performance metrics from NSProcessInfo are cross-validated against system APIs, while network module transmissions via NSURLSession are stress-tested under simulated low-bandwidth or intermittent connectivity.

Performance and regression testing address potential overhead and systemic risks. A benchmark suite compares application behavior with and without telemetry enabled, profiling CPU, memory, and storage usage to quantify impacts on resource-constrained devices (e.g., older iOS hardware). Stress tests simulate high-frequency user interactions to ensure the telemetry queue and background threads handle load without UI jitter or latency spikes. Regression tests rerun GNUstep's full test suite to detect unintended side effects, with targeted checks on features prone to interference, such as NSUserDefaults (telemetry configuration) and **NSNotificationCenter** (event distribution). UI testing complements this by monitoring frame rates and input responsiveness during peak telemetry activity, ensuring animations and user inputs remain smooth.

Test environments are staged to mirror real-world conditions. Initial development environments use debug builds with instrumentation to trace telemetry data flow and identify leaks or race conditions. Staging environments replicate production setups, deploying the enhanced GNUstep to collect metrics under realistic workloads without exposing end users to instability. Network condition simulators simulate scenarios like 3G latency or packet loss to validate the robustness of the retry mechanism and offline queue. Post-deployment, monitoring and reporting mechanisms track operational health: telemetry-specific logs record transmission errors, while performance dashboards highlight deviations in CPU/memory trends. User feedback from staged rollouts identifies subjective issues, ensuring the enhancement aligns with both technical and experiential requirements.

This comprehensive approach ensures the telemetry subsystem operates reliably across diverse scenarios while preserving GNUstep's core functionality and user experience.

12. Potential risks

The implementation of automatic metrics collection and transmission capabilities within the GNUstep system introduces several potential risks such as security, privacy and Maintainability.

Security Risks:Transmission of usage metrics over networks creates potential attack vectors for unauthorized data interception through man-in-the-middle attacks.**Impact**: Attackers could capture sensitive usage patterns or application state information, potentially revealing user behaviors or application vulnerabilities.

Privacy Risk: Without proper controls, the metrics system might inadvertently collect personally identifiable information (PII) or sensitive data.**Impact**: Violation of user privacy expectations and potential non-compliance with data protection regulations.

Maintainability Risk: Metrics collection functionality spans multiple architectural layers, potentially violating separation of concerns if not properly designed.**Impact**: Increased complexity of the codebase, making future modifications more difficult and error-prone.

13. Limitations & Lessons Learned

Architectural Limitations and Technical Challenges

While the telemetry system brought in some great features, it wasn't without its issues. Using method swizzling made the system fragile—any changes in GNUstep's core could break things. Depending heavily on SQLite also turned out to be tricky, especially for devices with limited storage or slower hardware. There were also concerns about user privacy, especially with unencrypted local files, and testing event hooks in the GUI proved harder than expected. Plus, because telemetry had to reach into multiple layers, keeping a clean separation between parts of the system wasn't always possible.

Collaborative Limitations

While the collaboration among team members was productive, several limitations impacted the group's efficiency. Despite having clearly divided responsibilities, external time constraints and conflicting academic commitments occasionally delayed progress and reduced availability for synchronous work. A major challenge stemmed from differing levels of familiarity with the existing GNUstep architecture and the tools used, particularly Understand. Some members were more actively engaged in the technical analysis, while others focused on peripheral tasks, leading to gaps in shared understanding. This disparity complicated efforts to evaluate and propose the two implementation strategies, as members interpreted the system's components and architectural constraints differently. Communication issues, including delayed responses and unclear documentation of decisions, further slowed consensus-building and led to duplicated efforts in certain sections. While the group made meaningful progress, these collaborative limitations highlight the need for more structured communication, better knowledge-sharing practices, and regular check-ins to ensure alignment in future team projects.

14. Reference

1.*IBM. (2025, February 14). TLS 1.3 protocol. IBM Documentation. Retrieved April,4 2025, from https://www.ibm.com/docs/en/sdk-java-technology/8?topic=handshake-tls-13-protocol*

2. GNUstep. (n.d.). Foundation. GNUstep MediaWiki. Retrieved April,4 2025, from https://mediawiki.gnustep.org/index.php/Foundation