# GNUSTEP ENHANCEMENT PROPOSAL: LLM SUPPORT

Group 12 – CISC/CMPE 322/326 – Queen's University

Daniel Tian
21dt41@queensu.ca
Samuel Tian
21st114@queensu.ca
James Choi
19jc132@queensu.ca
Christian Pierobon
christian.pierobon@queensu.ca
Luca Spermezan
22ls18@queensu.ca
Andrew Bissada
21ajb37@queensu.ca

## Abstract

The enhancement proposed in this report aims to add support for integration with large language models (LLMs) in GNUstep. The motivation for this enhancement involves the growing usage of LLMs in software applications, and the benefits of such features to GNUstep users and developers. This enhancement will provide access to various services provided by external LLMs, as well as a simplified interface for developers to integrate LLM services into their applications. The effects and potential risks of the enhancement involve maintainability, evolvability, testability, performance, and scalability. A testing plan for the enhancement would involve integration, regression, security, and performance testing. The implementation of this enhancement would introduce a new *LLM Support* component to the Application layer of GNUstep, which would be responsible for calling the external LLM API and handling events that need LLM services. Two approaches for implementation are integration in the responder chain, or integration in the notification subsystem. The major stakeholders of this enhancement are GNUstep contributors, developers, and users. The major non-functional requirements (NFRs) associated with these stakeholders are maintainability, evolvability, scalability, security, and performance, specifically response time and request load. The pros of the responder chain approach are good maintainability and consistent response times, while the cons are weak evolvability and low scalability. The pros of the notification subsystem approach are good evolvability and high scalability, while the cons are lesser maintainability, more inconsistent response times, and higher potential for security risks. In the end, implementation approach 2 using notification observers was chosen as the optimal implementation for this enhancement. Two use cases for this enhancement involve a developer connecting a TTS LLM service to a button in a GUI application, and a user pressing such a button to get the TTS output.

# Introduction

## Purpose

The purpose of this report is to present a proposal to add Large Language Model (LLM) Support as an enhancement to GNUstep. This report provides a detailed breakdown of the enhancement features, along with a discussion of the associated impacts and risks. An in-depth SEI SAAM analysis is also included, comparing two implementation approaches. This report is useful to GNUstep contributors looking to add LLM support to GNUstep, as it provides a comprehensive breakdown of impacts on the high-level system architecture.

## Organization

This report is divided into six major sections. The first section is the introduction, which summarizes the main report contents. The second section provides a detailed overview of the LLM support enhancement. The third section presents impacts and risks associated with the enhancement on GNUstep. The fourth section discusses implementation details and presents two possible approaches. The fifth section conducts SEI SAAM analysis on these approaches. The sixth section presents the major conclusions and lessons learned.

## Conclusions

The addition of LLM support as an enhancement to GNUstep would provide functionalities that keep it relevant amidst the growing dominance of LLMs in software. Maintainability and evolvability are key considerations to consider, and thorough documentation would be needed to supplement these considerations, alongside testing of multiple requirements. Implementation should be conducted with performance and scalability in mind to avoid introducing bottlenecks or a single point of failure. The implementation approach using notification observers is recommended, bearing major benefits and low-risk shortcomings. A key lesson learned was the importance of re-using pre-existing architectural structures when adding enhancement, which is applicable to both this, and future enhancements.

# Enhancement Overview

## Motivation

The addition of LLM support to GNUstep would be a game changer for windows-based GUI development in a world where AI is being adopted and integrated at breakneck speed across all industries. The addition of LLM support to GNUstep would allow GUI developers to take advantage of the powerful generative capabilities of LLMs in their applications, and would provide novel features for users to interact with. It would also greatly improve the competitiveness of windows-based applications in the market, thus increasing GNUstep's

standing amongst its competitors. This enhancement would also provide GNUstep with a native solution for LLM integration, streamlining the process for application developers and providing a more intuitive and understandable interface for using LLM services.

## Features

This LLM support enhancement would provide GNUstep applications with many features provided by external LLMs. This would include services from various models, such as text, image, and audio generation.[3] In addition, it would also support integration with a diverse range of LLMs, such as ChatGPT and Llama. The services provided by these LLMs would be available for developers to add to their applications, and for users to interact with and use. In addition, this new enhancement would provide developers with a stable and easy-to-understand interface for adding LLM services to their GUI applications. Instead of directly calling the LLM API themselves, developers could simply use the interfaces provided by this enhancement, which would handle all API calls and updates on the developer's behalf. This would greatly simplify LLM integration by eliminating the need for constantly researching and updating API calls, while also reducing potential for bugs by human error.

# System Considerations

## Effects of Changes

### Maintainability

The addition of LLM support to GNUstep would increase system maintenance complexity, as its behavior is not natively supported by the Apple Cocoa framework.[1] As a result, developers familiar with Cocoa may have a difficult time maintaining these new features, as its documentation would no longer be a reliable reference. Thus, extra documentation would be needed to ensure the integration logic remains clear. In addition, maintenance of new LLM support components would need to be added to existing maintenance routines.

### Evolvability

The LLM Support component evolves in concordance with the external LLM APIs it calls. LLM APIs evolve rapidly, and developers would need to ensure that the enhancement remains up to date and compatible.[3] Authentication mechanisms, API parameter formats, or rate limiting behaviors would need to be updated frequently, meaning this component would be rather unstable. This could lead to major increases in system evolution costs.

### Testability

A primary concern involved with testing LLM support is the non-deterministic nature of LLM output. Unlike normal functions and methods which have deterministic and predictable

behaviours, LLMs can provide different outputs for the same input based on temperature settings, prompt phrasing, or backend model updates.[3] As such, automatic testing would be difficult, as the expected results of tests must be defined based on broad context, and output interpretation would need to be more nuanced compared to traditional methods.

### Performance

By depending on external LLMs for services, system performance, specifically response times, could be impacted, as API calls to LLMs are typically network-bound, and can be computationally taxing to the service provider.[3] This delay in response time could degrade user experience, especially if quick feedback to actions is expected, such as for text captioning. As a result, measures for optimizing responsiveness would be needed, such as asynchronous request handling, which could add to the architectural complexity.

## Potential Risks

### Security

Being an external interface, there are several security risks associated with connecting to external LLMs through their API. LLMs handle vast loads of data, with many containing personal user information.[3] Thus, the transmission and storage of this sensitive data must be done securely to prevent data leaks. This would require strict data retention principles and the use of proper encryption methods. Interception of data in transit is another security risk that could result in the injection of malicious data, data corruption, or damage to the internal system. Mitigating such a risk would require robust authentication and encryption systems that could verify senders and receivers. Finally, API keys would need to be stored securely to prevent them from being stolen or held hostage for ransom.[3]

### Maintainability

The major maintainability risk involved with this enhancement is the dependence on external API updates and changes. The companies that run LLMs are liable to update the requirements or functionality of their APIs at any time, meaning the GNUstep team would need to always be ready to update their own components that depend on the API to ensure the enhancement features remain up-to-date. Given that GNUstep is only a small project with limited developers, this requirement would become a great burden, and could cause the project to quickly fall behind–or fail altogether–if the API were to change too quickly.

### Performance

The major performance risk introduced by this enhancement involves the fact that LLM uptimes and responses are completely out of the control of GNUstep. This means that if the servers hosting the LLM were slow in their response times, the functionalities provided

by this enhancement would suffer from poor performance. Worse even, if the LLM servers were to go down without warning, the functionalities provided by this enhancement would become unavailable. Thus, this enhancement makes GNUstep liable to failure of external LLMs, which is completely uncontrollable from within the GNUstep system itself.

### Testing Plan

The testing plan for this enhancement would involve multiple specialized tests. Integration tests would be needed to validate the correctness of the interactions between GNUstep and the external LLMs, ensuring proper data exchange and response handling. Non-determinism in LLM output should also be accounted for. Once implemented, regression testing should be conducted regularly after updates to identify unwanted side effects, and could be automated for minor API updates to reduce developer burden. Security testing would also be needed to identify potential vulnerabilities, and performance testing would be needed to verify API response times and maximum request load for new features.

# Implementation

## New Components

### LLM Support

This is a new component that will be added to the Application layer of GNUstep. It is responsible for handling events that need LLM services, calling the API for external LLMs, and managing the input, output, and logging of such processes. This new component will provide LLM Controller classes for handling events in this way. When a developer wants to attach some LLM service to a GUI element, they will create a new LLM Controller, which will be responsible for handling API calls for that specific GUI element. In this way, the functionalities provided by this component can be cleanly encapsulated within one controller for each GUI element, conforming to the object-oriented style of GNUstep.[2] In addition, it will allow for easier maintenance and evolution, as LLM Controllers can be easily swapped in and out, with no features left behind scattered throughout the system.

## Component Interactions

### Base Functionalities

Similar to all other components in the Application layer, the *LLM Support* component will depend on both the *Runtime Utilities* and the *Value Data* sub-components in the *Base Functionalities* component. The *Runtime Utilities* sub-component provides fundamental, basic services, such as error handling, logging, and task management. The *Value Data* sub-

component provides basic data structures, such as dictionaries and arrays, for storing parameters for the API calls, and defines how output returned by API calls is represented.

## OS Services

Similar to all other components in the Application layer, the *LLM Support* component will depend on the *OS Services* component. This component provides various crucial low-level functionalities, such as networking, inter-process communication, and file management, for implementing the operations performed by LLM Controllers. HTTP Requests are also handled by this component, which allow LLM Controllers to call external LLM web API.

## Back-end

Similar to all other components in the Application Layer, the *LLM Support* component will depend on the *Back-end* component, as well as its internal sub-components. The *LLM Support* component will depend on the *Main Application Environment*, which will provide connections to external interfaces the LLM Controllers may need to access. In addition, the *Main Application Environment* will also depend on the *LLM Support* component, as it is responsible for tracking and managing all active controllers in the system, and is the only component able to create and destroy LLM Controllers. This is to address security risks, as the management of LLM Controllers will become centralized in this component, making it easier to track controller activities and identify unverified connections outside the *Main Application Environment*. The *LLM Support* component also depends on the other sub-components in the *Back-end*, such as *Drawing*, *Text*, and *Audio*, to manage LLM output.

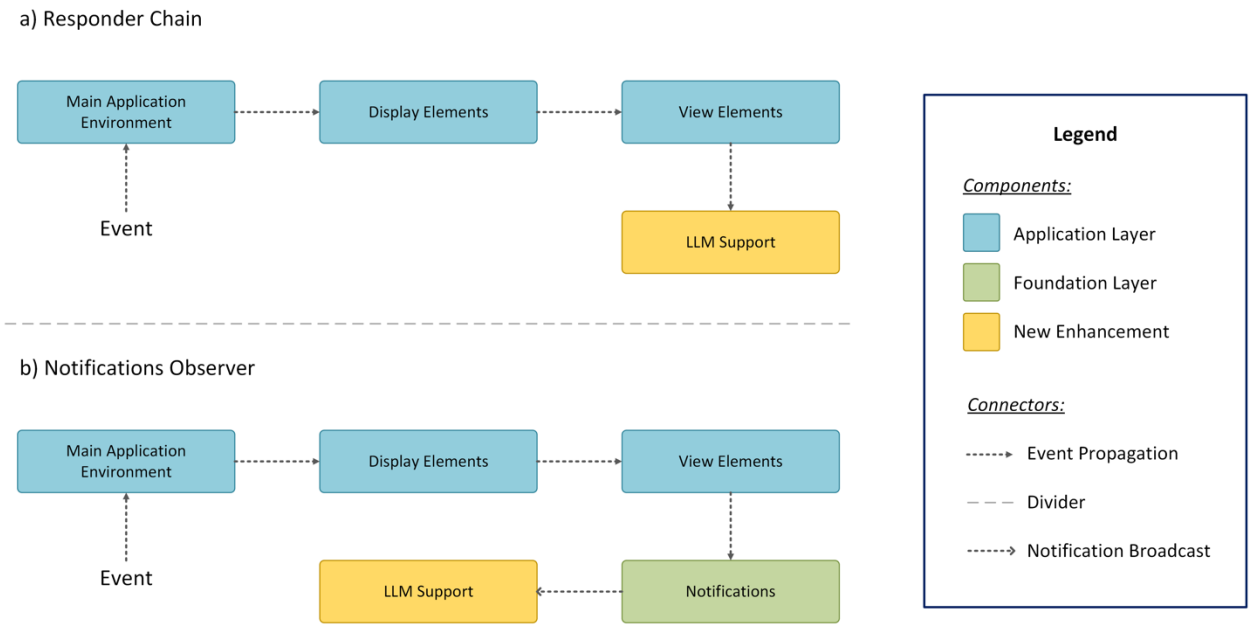



**Figure 1.** Comparison of the two implementation approaches

## Approach 1: Responder Chain Integration

### Architectural Changes

The first implementation approach for this enhancement integrates the *LLM Support* component into the GNUstep responder chain, which would introduce a new two-way dependency between the *View Elements* and the *LLM Support* components. This involves layered and object-oriented style. In this approach, GUI elements can propagate events directly to LLM Controllers through the chain, which respond by calling the LLM API.

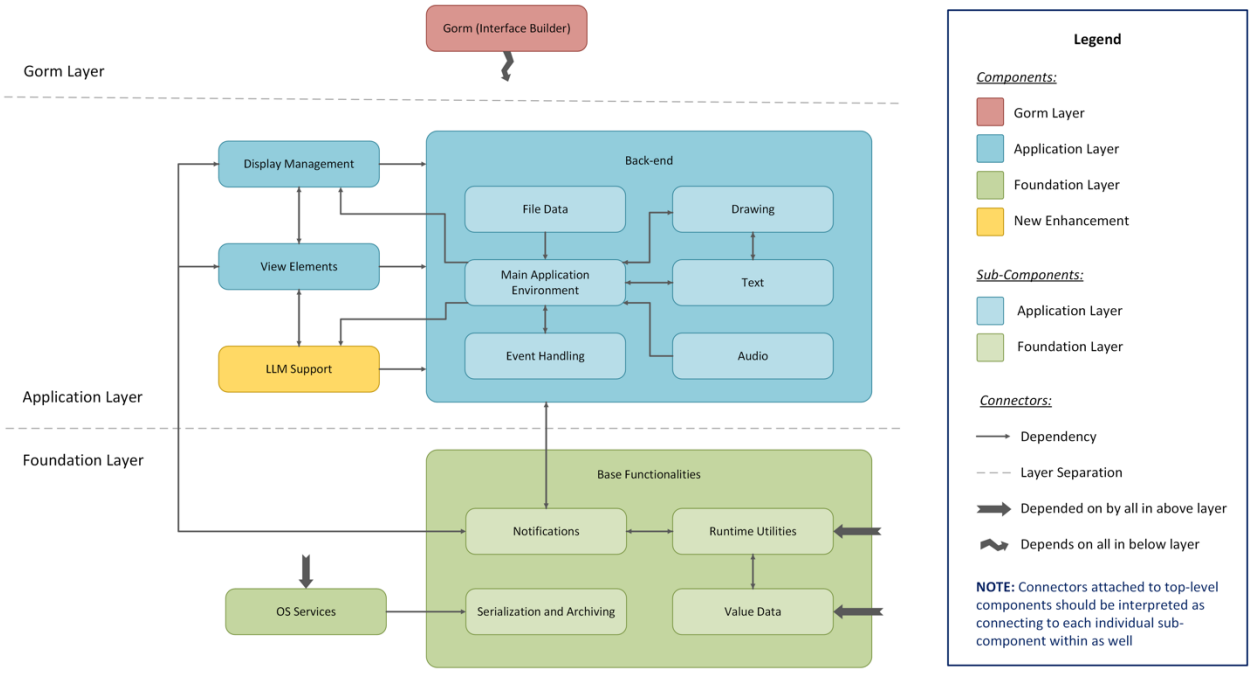



**Figure 2.** Conceptual architecture for implementation approach 1

### Directories and Files

The main library that will be affected by this enhancement will be the *libs-gui* library in GNUstep.[2] New files, such as those containing LLM Controller classes, would need to be added to the library. In addition, several pre-existing files would need to be modified as well, such as the Storyboard classes in the *Main Application Environment*, which track all active controllers in a running application.[1] The *Event Handling* classes would also need to be modified to provide services to LLM Controllers. For this implementation approach specifically, the *View Element* classes would also need to be modified to propagate events to LLM Controllers. This would require changes to related event dispatch methods.

## Approach 2: Notification Observers

### Architectural Changes

The second implementation approach for this enhancement connects the *LLM Support* with the notification subsystem, which would introduce a new two-way dependency between the *Notifications* and the *LLM Support* components. This involves layered and object-oriented style, with features of publish-subscribe. In this approach, the LLM Controllers register as observers to notification centers. Then, when GUI elements receive events that need LLM services, they do not propagate events directly to LLM Controllers, but instead pass them to notification centers, which broadcast the events to observers. The LLM Controllers then receive the notifications, and respond by calling the LLM API.

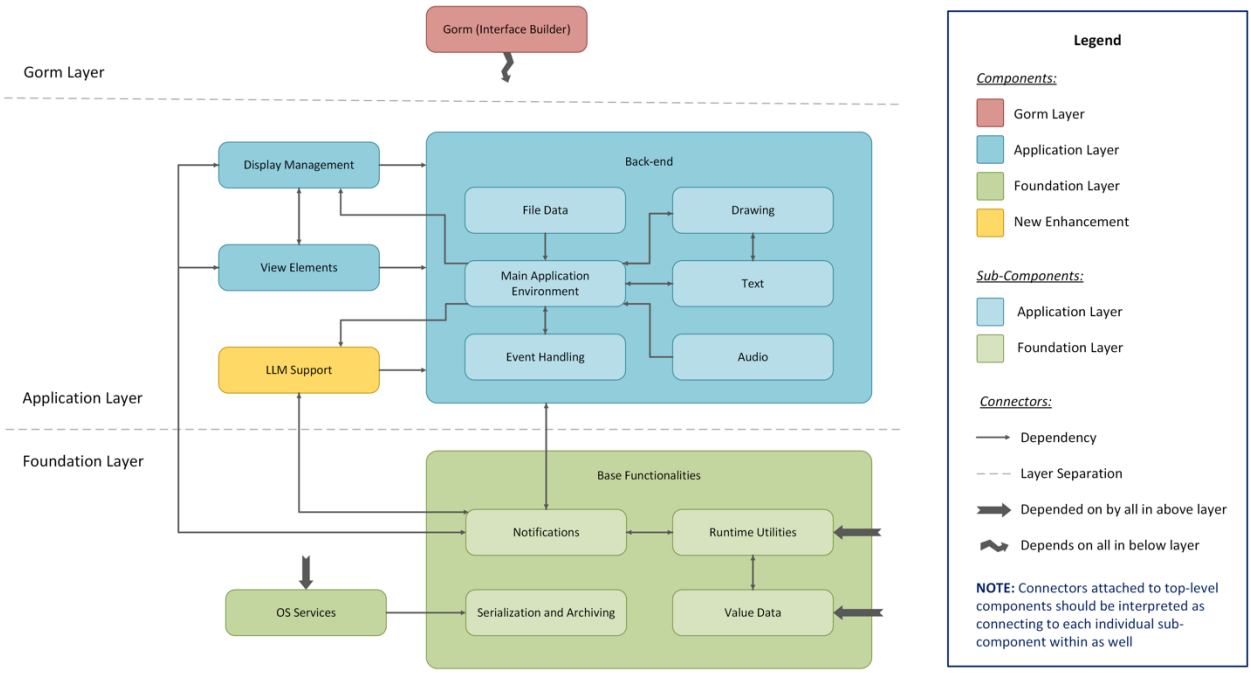



**Figure 3.** Conceptual architecture for implementation approach 2

### Directories and Files

Similar to implementation approach 1, the main library that will be affected by this enhancement will be the *libs-gui* library.[2] New classes, as well as the changes to the *Back-end* component files, would be similar to those described in approach 1. However, for this approach, the *Notifications* classes in the *libs-base* library would also need to be modified to recognize LLM Controllers as valid observers.[2] Some new notification classes would also need to be added to define new data types and operations from LLM API calls.

# SEI SAAM Architectural Analysis

## Major Stakeholders

The first major stakeholders affected by this enhancement are the GNUstep contributors, who are responsible for developing the GNUstep framework and ensuring the stability, modularity, and long-term viability of its components. They would be directly affected by this enhancement as the introduction of the *LLM Support* component into the system architecture would give them a new feature that they need to maintain, debug, and evolve.

Another major group of stakeholders is GNUstep developers who build GUI applications through Gorm. These developers would be among the first to adopt and utilize the *LLM Support* functionalities in their applications, as they would be using GNUstep and Gorm frequently for either their employment activities or personal projects.

The final major stakeholders are the end-users of GNUstep applications. These users would directly interact with the LLM services provided by this enhancement, such as text-to-speech or text generation. Thus, their experience with the services provided by this enhancement would be heavily impacted by the implementation of the new component.

## Major Non-Functional Requirements

For GNUstep contributors, the most important non-functional requirements (NFRs) are maintainability and evolvability. Maintainability would be important, as they would want the *LLM Support* component to be easy to modify or debug when issues arise, ensuring that maintenance costs do not increase substantially. Evolvability is especially critical, as the landscape of LLM technology is changing rapidly, with new APIs, features, and best practices emerging on a regular basis. Thus, GNUstep contributors would want this new component to be implemented in a way with minimal coupling, allowing it to evolve independently from other core components, and reducing side effects from updates.

For GNUstep and Gorm developers, the primary NFRs are scalability and performance, specifically with respect to request load. Scalability and tolerance to high request loads would be important to these developers, as they would want their GUI applications to remain functional even under high-traffic conditions. If the new features are not scalable, developers would likely choose not to use them to ensure a reliable application uptime. Ease-of-use is another NFR developers would be concerned with, as a feature that is too difficult to understand would be unappealing for them to learn how to use.

For GNUstep users, the major NFRs are security and performance, specifically with respect to response time. Security is important, as users often need to input sensitive information, and trust the system that their data will be handled appropriately. Thus, if they

realize that new features are unsafe, they may be unwilling to use them. Response time is important as well, as users expect quick and reliable responses from interactive GUI applications, and any noticeable lag or inconsistency can lead to frustration.

## Implementation Evaluation

From a maintainability perspective, approach 1 is highly maintainable, as it integrates directly in the GNUstep responder chain. This means that GNUstep contributors with pre-existing knowledge about the responder chain would be able to easily understand how the new component fits and interacts with the rest of the system, making it easier for them to identify and address potential bugs. In addition, previously established methods used in the past for maintaining the responder chain could be easily adapted and extended to account for the new component. In contrast, approach 2 is more difficult to maintain, as notification centers can send many notifications at a time to a wide range of observers. This would make it extremely difficult to trace specific notifications sent to the *LLM Support* component, thus increasing the complexity of bug detection and correction.

From an evolvability perspective, approach 1 is not easily evolvable. This is because the integration of *LLM Support* in the responder chain tightly couples the *View Elements* and the *LLM Support* components through a two-way dependency. This is problematic due to the rapidly evolving nature of LLMs, as APIs have the potential to change quite frequently. This means that the *LLM Support* would need to evolve alongside the API, and due to the tight coupling with the *View Elements* component, the *View Elements* would also need to evolve alongside the *LLM Support* as well. This would greatly increase system evolution costs, and would unnecessarily destabilize the *View Elements* component. On the other hand, approach 2 is much more evolvable. Although there is a two-way dependency between the *LLM Support* and *Notifications* components, this coupling is much more manageable. This is because in approach 2, other components do not need to know what events the *LLM Support* can handle. When *View Elements* receive an event they are unable to handle, unlike in approach 1, they do not need to decide whether to propagate the event to an LLM Controller or some other handler. Instead, they can unconditionally send the event to the *Notifications* component, which itself broadcasts the event to all observers without needing to know what services they provide. This ensures that these component interactions remain stable, even if the *LLM Support* features undergo significant changes.

From a performance perspective, both approaches have their respective benefits and shortcomings. In terms of response time, approach 1 is highly consistent and reliable. This is because the event propagation through the responder chain guarantees a response from the next responder, which keeps response times consistent. In addition, message passing to direct named targets means that responses can easily be tracked and verified. On the other hand, approach 2 does not always guarantee a response, as notifications are sent to

all observers without guarantee of a return, making response times more variable and unpredictable. However, in terms of request load and scalability, approach 2 may be more scalable. This is because notifications can be sent asynchronously to all observers without the need for choosing a direct target. This would ensure that a backlog of events in the notification centers does not build up under high loads. In addition, if LLM Controllers implement their own queues for receiving notifications, the burden of the request load will be distributed across a wide range of LLM Controllers, rather than being condensed in the *Notifications* component. In contrast, approach 1 would be much less scalable with high request loads, as *View Elements* would need to choose propagation targets for each event coming in. If these operations are synchronous, this could cause further delays under high request loads, and the *View Elements* component could become a major bottleneck.

From a security perspective, both of these approaches are reasonable, as security risks normally involve the actual message passing mechanisms themselves, rather than the structures in which they operate. However, it should be noted that approach 2 may have a higher risk of data leaks, as an unverified entity could register as an observer during a security breach, allowing it to receive unauthorized notifications. This attacker would also be difficult to track down and eliminate due to the nature of asynchronous notifications.

## Optimal Implementation

The optimal implementation for this enhancement is approach 2. This is because this approach provides significant benefits, while also having less detrimental shortcomings.

In terms of maintainability, even though approach 2 is less maintainable than approach 1, there are measures that can be taken for improvement. For example, notification logs can be kept, making it easier to track broadcasts and responses. This may incur extra overhead during system operation, but if it cuts down on maintenance costs, it may be worth it. In addition, integration with the notification subsystem means that pre-existing maintenance methods for notification centers may still be re-usable to an extent.

In terms of evolvability, the extra costs in approach 1 associated with evolving the *View Elements* component may be too much for developers to bear. If multiple changes to the LLM API were to occur in rapid succession, developers may not be able to keep up with the extra evolution of the *View Elements*. Over time, project plans would become delayed, which would take time away from bug fixes and work on other components. In the long run, either shortcuts would need to be taken, leading to the degradation of the architecture, or abandonment of the project altogether. Thus, approach 2 is much safer in this regard.

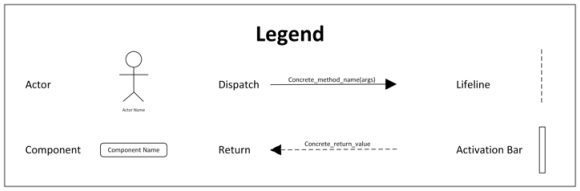
In terms of performance, approach 2 addresses the more important issues surrounding application availability. From both a developer and user perspective, limited functionality is always better than no functionality at all. Thus, even though approach 1 may present

more consistent and predictable response times, approach 2 would be preferred, as it ensures that the system will not fail under high request loads. This makes approach 2 favourable for the major stakeholders who care about reliable uptime and availability.

In terms of security, although approach 2 had a noticeable vulnerability, as long as proper security measures, such as encryption and firewalls, are put in place, the actual risk of this vulnerability is likely quite low. In addition, approach 1 is not without vulnerabilities itself, as its low scalability may make it prone to DDoS attacks. Thus, with the proper security measures put in place, approach 2 is a viable approach in terms of system security.

# Use Cases

This section will cover two use cases for users and stakeholders that interact with the new *LLM Support*. The provided legend (right) describes both sequence diagrams.




## Use Case 1: Connect Button to Text-to-Speech LLM Service

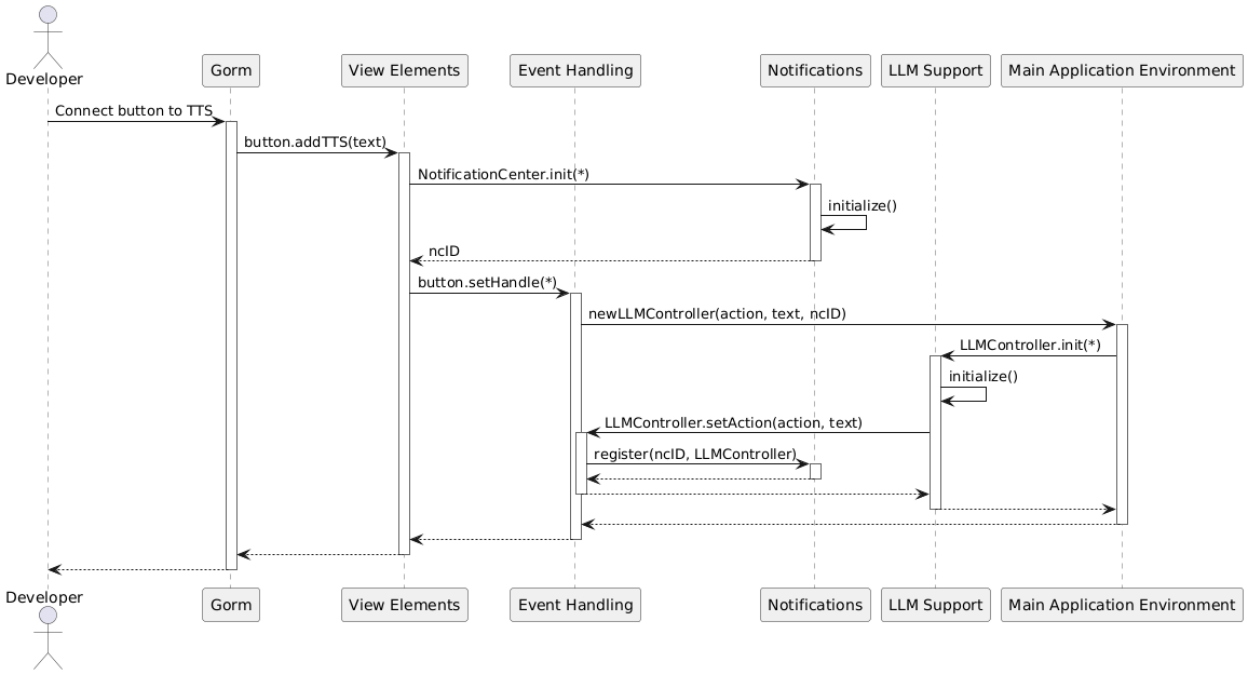



**Figure 4.** Sequence diagram for use case 1

The sequence diagram in Figure 4 illustrates the following use case: “a developer using Gorm connects a button to a text-to-speech feature provided by an LLM”. Text-to-speech (TTS) is a common service provided by LLMs through API calls, and this sequence diagram shows how such a service can be connected to a pre-existing button in a GUI application.

First, the developer interacts with *Gorm*, which is prompted to connect the button to a TTS feature. Gorm then calls the *View Elements* component responsible for managing the button. The *View Elements* component calls the *Notifications* component to create a new Notification Center object, which will be used to broadcast the button events to LLM Controllers, and the ID of the newly created Notification Center is returned.

Next, the *View Elements* component calls the *Event Handling* component to determine how events from the button will be handled. The *Event Handling* component does this by calling the *Main Application Environment* to create a new LLM Controller, which in turn calls the *LLM Support* component to do so. This LLM Controller then calls the *Event Handling* component to set its own action for handling events, and the *Event Handling* component registers the LLM Controller to the newly created Notification Center. It does this by using the ID for the Notification Center corresponding to the button.

## Use Case 2: Text-to-Speech

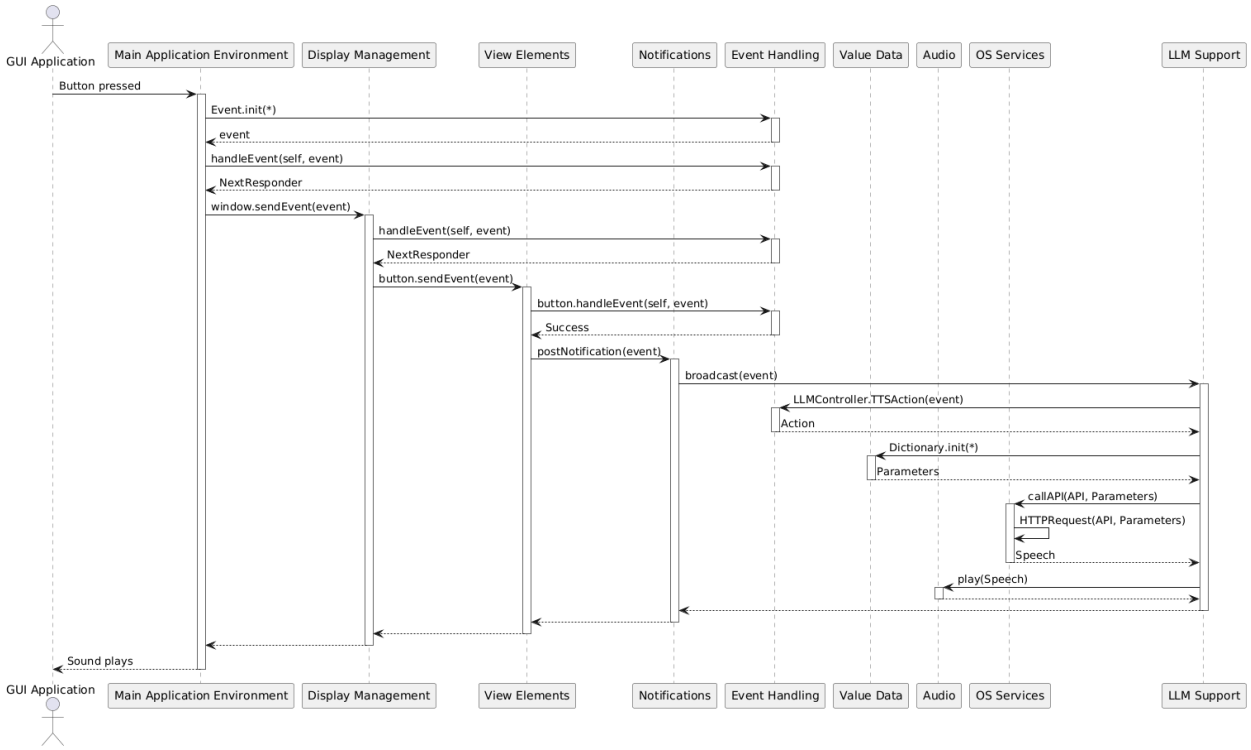



**Figure 5.** Sequence diagram for use case 2

The sequence diagram in Figure 5 illustrates the following use case: “a button is pressed, which plays a text-to-speech narration of some text, provided by an LLM”. The first part of this sequence diagram represents the responder chain in GNUstep. The *Main Application Environment* first creates an Event object, which is propagated down the responder chain to the *Display Management*, and then the *View Elements* component. At each step, each

component calls the *Event Handling* component, which determines whether the current component should handle the event, or pass it down the chain to the next responder.

The second part of this sequence diagram shows how the event is handled. The *View Elements* component sends the event to the *Notifications* component, which broadcasts the event to the LLM Controller in the *LLM Support* component. This controller then calls the *Event Handling* component for an action in response. This action prompts the LLM Controller to store the parameters for an API call in a Dictionary structure from the *Value Data* component. The LLM Controller then calls the necessary API for TTS through an HTTP Request from the *OS Services* component. When the API returns, the LLM Controller calls the *Audio* component to play the speech from the TTS output for the user to hear.

# Conclusions & Lessons Learned

This report has presented LLM support as a new enhancement to GNUstep. The motivation for this enhancement stems from the recent rapid growth in the relevance of LLMs, which are becoming increasingly important in various application domains. This enhancement would allow GNUstep to integrate with external LLMs through web APIs, providing LLM services that can be added to GUI applications, such as text or image generation.

System maintainability is a crucial consideration for the implementation of these new features. In particular, past maintenance procedures may be inapplicable to this new *LLM Support* component, especially if they have been heavily reliant on the Apple Cocoa documentation. System evolvability should be considered as well, as GNUstep would need to evolve alongside the external LLM services to ensure that changes to the API are accurately and swiftly reflected in the GNUstep system to avoid becoming outdated.

The non-deterministic nature of LLM output also presents important considerations for testing, as traditional test case approaches would no longer be applicable. In this report, we have presented a testing plan that involves integration, regression, security, and performance tests. Security measures would also need to be included to ensure that issues surrounding user privacy, data integrity, and API keys are addressed properly.

In terms of performance and scalability, response time and request load are two major factors that should be monitored closely. If network connections are unstable or LLM servers are slow or down, access to the *LLM Support* services could be lost completely. Connections with these external interfaces should be designed with speed and reliability in mind to ensure they do not become a bottleneck or a single point of failure in the system.

Out of the two implementation approaches presented in this report, approach 2 using notification observers is the one we recommend. This approach provides an easily

evolvable and highly scalable implementation of the *LLM support*, while still providing a relatively maintainable and responsive set of features. Approach 1 is not advised, as it has critical flaws in evolvability and scalability that could become major issues to the project.

A key lesson learned from this study was the benefit of re-using pre-existing architectural styles and design patterns. By directly integrating our enhancement into the pre-existing notifications subsystem in GNUstep, integration with the rest of the system became much more intuitive. Had we chosen to use a new design pattern or style, integration complexity would have increased significantly, leading to a higher risk of potential bugs or oversights in development. Thus, for future evolutions of GNUstep, developers should always first try to reuse the existing architecture to minimize associated costs and potential for mistakes.

# Data Dictionary

- **GNUstep Contributors:** Developers who work on the GNUstep source code.
- **GNUstep Developers:** GUI Developers who use GNUstep to build applications.
- **GNUstep Users:** People who use and interact with GNUstep applications.
- **Large Language Model:** A class of machine learning model that is used to perform natural language processing tasks, such as generating text, image, or audio data.
- **Application Programming Interface:** An interface provided by a system that allows other systems to communicate with it and use its services.
- **Regression Testing:** Testing method that checks for side effects in new updates.
- **Text-to-Speech:** A feature provided by various LLMs that takes text data as input and produces audio as output, where the audio is a spoken narration of the text.

# Naming Conventions

- **LLM:** Large Language Model
- **GUI:** Graphical User Interface
- **TTS:** Text-To-Speech
- **HTTP:** Hypertext Transfer Protocol
- **Gorm:** Graphical Object relationship modeler
- **API:** Application Programming Interface
- **DDoS:** Distributed-Denial-of-Service
- **NFR:** Non-Functional Requirement
- **SEI SAAM:** Software-Engineering-Institute Software-Architecture-Analysis Method

# References


[1] Apple Developer Documentation

[2] GNUstep System Overview

[3] OpenAI Developer Platform

[4] Apple Cocoa Documentation