KODI Architectural Enhancement Report

CISC 322

Tuesday, December 5th, 2023

Alexander Muglia - 19am107@queensu.ca - 20208942

Ben Falkner - 21bmf2@queensu.ca - 20306606

Colin Cockburn - 20clc1@queensu.ca - 20238677

Daniel lister - 20dl51@queensu.ca - 20290719

George Salib - 20gs36@queensu.ca - 20281662

Julian Brickman - 19jmb10@queensu.ca - 20206218

TABLE OF CONTENTS

ENHANCEMENT = TAILORED CONTENT SUGGESTIONS (RECOMMENDED CONTENT)

ABSTRACT

In this report, we propose and analyze an enhancement to the Kodi media platform, focusing on introducing tailored content suggestions to improve user experience. The enhancement is explored through two approaches: a client-side recommendation system that suggests content based on user interactions, and a server-side algorithm that leverages broader data for more comprehensive recommendations. We evaluate the impact of these approaches on Kodi's architecture, considering additions like the Suggestion Algorithm node and content aggregation servers, and assess their effects on maintainability, evolvability, testability, and performance. A SAAM analysis is included to understand the implications for different stakeholders such as Kodi developers, plugin developers, and users, with a focus on non-functional requirements like performance and security. The report also outlines a detailed testing plan encompassing performance, user experience, and security aspects to ensure the enhancement's effectiveness. Overall, this report provides a balanced analysis of the proposed enhancements, their architectural integration, and the potential benefits and challenges they present.

INTRODUCTION & OVERVIEW

The Kodi media platform, renowned for its flexibility and extensive customization options, stands at the forefront of media center software. However, as user demands evolve and the digital content landscape expands, there is a compelling need for Kodi to enhance its user experience by incorporating more personalized features. This report delves into a significant proposed enhancement for Kodi: the introduction of tailored content suggestions, also known as recommended content. This enhancement aims to elevate the user experience by providing more relevant and engaging content suggestions based on individual user preferences and interaction history.

The report is structured to provide a comprehensive understanding of this enhancement, covering its conceptualization, implementation approaches, and potential impacts on the Kodi system. We begin by exploring two distinct approaches for the recommendation system: a client-side implementation and a server-side implementation. The client-side model focuses on a localized recommendation algorithm that operates based on the user's current interactions with the content. In contrast, the server-side model employs a more holistic approach, using a backend algorithm that gathers and analyzes data across a broader spectrum of user interactions.

We then proceed to examine the implications of these enhancements on Kodi's existing architecture. This includes an analysis of the necessary modifications to the system architecture, detailing the integration of new components such as the Suggestion Algorithm node and content

aggregation servers. The report also assesses how these enhancements influence key factors like system maintainability, evolvability, testability, and overall performance.

A crucial part of this report is the SAAM (Software Architecture Analysis Method) analysis, which offers insights into the impact of the proposed enhancements on various stakeholders, including Kodi developers, plugin developers, and end-users. This analysis helps us understand the prioritization of non-functional requirements such as performance, maintainability, testability, and security, which are essential for the successful implementation of the recommendation system.

Finally, we present a detailed plan for testing these enhancements, outlining strategies for performance testing, user experience testing, and security testing. This plan is designed to ensure that the proposed enhancements not only integrate seamlessly with the existing Kodi architecture but also meet the high standards of functionality and reliability expected by its users.

In summary, this report provides an in-depth exploration of the proposed enhancement for Kodi, offering a detailed overview of its implementation approaches, architectural implications, stakeholder impact, and testing strategies. Our objective is to present a well-rounded analysis that guides the effective and efficient enhancement of the Kodi platform, ensuring it remains a top choice for media enthusiasts worldwide.

IMPLEMENTATION ARCHITECTURE DIAGRAMS

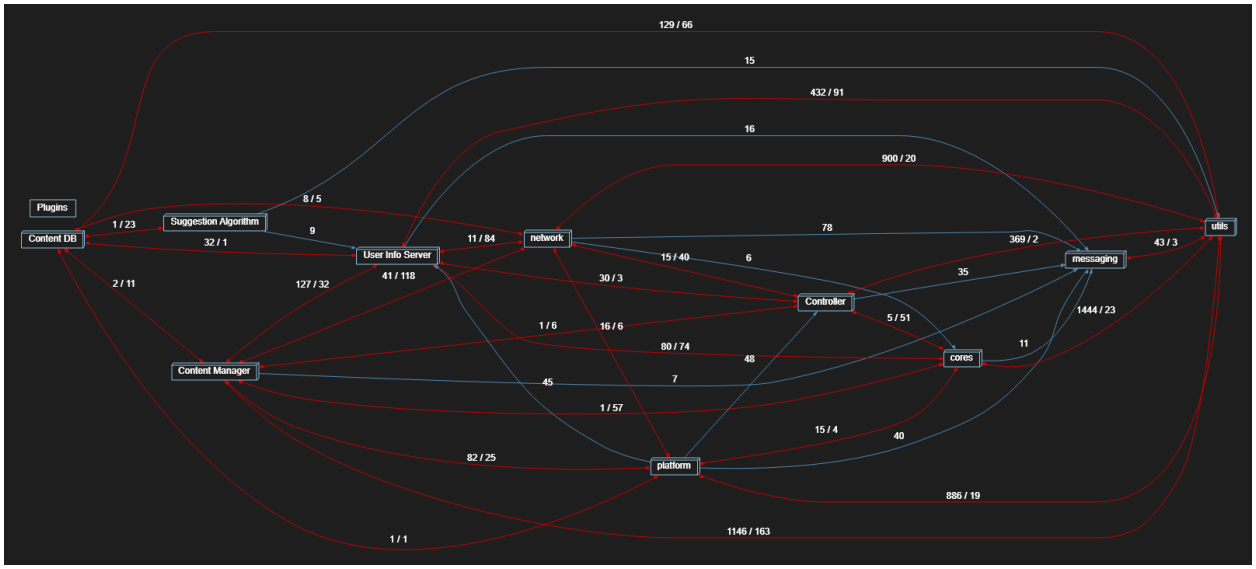

Figure 1: Client-Side Implementation Architecture Diagram

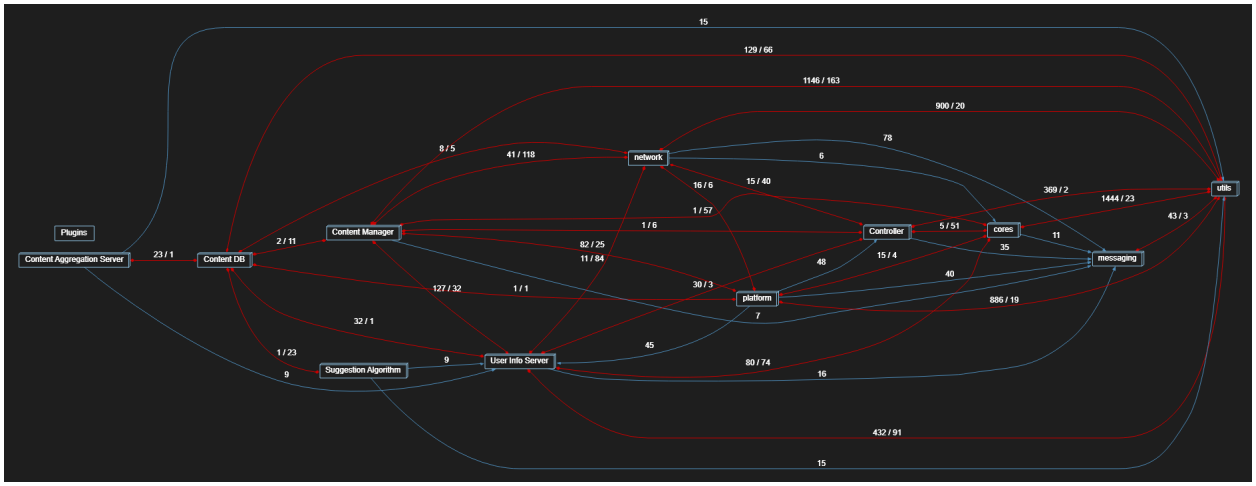

Figure 2: Server-Side Implementation Architecture Diagram

EFFECTS OF ENHANCEMENT / SYSTEM WITH ENHANCEMENT

Approach One (Client-Side):

The first approach of the proposed enhancement is the addition of a local client side recommendation system. This would contain a model trained on different types of content offered on KODI. As the user watches content, the algorithm generates recommendations based on the classifications of the content. For movies, this could mean what genre, year, or film style to suggest similar movies. Trained on datasets of current content from streaming platforms and online content databases, the algorithm would recommend content for the user to engage with. This feature is not a streaming service itself, only the recommendation. So the user would have to seek out the content separately following the recommendation.

Effects on system architecture:

The recommendation system would need the addition of a new Suggestion Algorithm node which is a node that communicates with the ContentDB node. Receiving information on the type of content the user is interacting with to generate new recommendations. This information then gets sent to the user info server and the Utils node for standard functions. The utils node would handle the display of the suggestions, as new GUI components would need to be made.

General effects on maintainability, evolvability, testability & performance:

KODI focuses on modularity. It uses the fact that it is open source to allow a huge amount of user created plugins to expand KODI's features. This means that KODI's core needs to be a very solid base for users to build upon. The standards for their core features are very high.

As such, the new proposed recommendation system would need to be able integrate seamlessly with different plugins. Our proposal would be to provide a testing framework for plugin developers. It would outline what types of information the algorithm takes as input. This streamlines the process of ensuring that a plugin integrates with the recommendation system. Since the new node for the algorithm is not meant to interfere with existing core functionality, it would not affect the current use cases of the system. Furthermore, since the recommendation algorithm is client side, the testing can become more focused on unit and functional tests. The downside to this implementation is that the recommendation model would need frequent updates, increasing the costs of maintenance.

Approach Two (Server-Side):

The second approach to implementing a recommendation system into the core KODI system is through a server side algorithm. This would contain a model trained on different types of content offered on KODI. Similar to the client side approach it would offer recommendations on what content to interact with next. This implementation would involve a backend with the model hosted allowing the user to connect through their network. The algorithm would be able to continuously evolve as the interaction with the recommended content can be returned. This would optimize the suggestions and improve the algorithm.

Effects on System Architecture:

The effects on the system architecture with the new added content suggestion feature adds two nodes to the core system. First the content aggregation server which is the connection to the backend of the content data the algorithm can suggest. Then the Suggestion Algorithm, which similarly to the client side implementation, generates the recommendations.

General effects on maintainability, evolvability, testability & performance:

As KODI evolves and gets new features, so does the amount of information that the recommendation algorithm receives. Thus, the algorithm would be tailored further and further towards your interests. Furthermore, the recommendation system could also learn to take in information across different entertainment forms. For example, the style of games you play could influence the genre of movies you get recommended. One downside to the server side implementation is that it would create a large amount of work in regards to maintenance. A server side database would need to be maintained with large amounts of up to date watch history of its users, in order to feed data to the recommendation algorithm. This all combines to mean higher expenses, as more maintenance means more workers you need to hire. Not to mention the costs of hosting a large server. Another big part effect of the server side recommendation system

is on testability. Extensive testing would need to be done to ensure that the system is so modular, even if servers are down or the user has no WIFI, KODI still runs flawlessly.

SAAM Analysis:

The following list contains the major stakeholders for the proposed recommendation system enhancement:

- KODI Developers
- Independent Plugin Developers
- Users

Each of these stakeholders has different interests in regards to the types of nonfunctional requirements they prioritize. The main types of NFR's that KODI's stakeholders care about are as follows:

Performance: The recommendation system should not have a large performance overhead
Maintainability: The recommendation system should be maintainable with minimal costs.
Testability: The recommendation system should be easily testable.
Security: The recommendation system should not expose any weaknesses to sensitive data, or core components.

KODI Developers care most about performance, maintainability and testability. They want to ensure that the recommendation system does not slow down other core components, is not too costly to maintain, and can be tested easily with the rest of the system.

Independent Plugin Developers care most about performance, maintainability and testability. They want to ensure that the recommendation system runs smoothly, is easy to keep up to date, and is highly testable with the new plugins they create.

Users care most about performance and security. They want to ensure that KODI runs quickly, does not lag, and is not vulnerable to hacks or data leaks that might expose personal information.

Comparing the two approaches:

On the one hand, a server side approach would be much more accurate. It would be consistently improving based on large amounts of data across numerous users of KODI. It would also be able to track your interests over a more extended period of time. Unfortunately this would require setting up a huge new part of the backend. Maintaining and hosting servers and

databases to keep track of all the user statistics would add significant costs. This does not align with KODI's client heavy architecture. The client side approach would be much more lightweight, and a lot easier to make modular. If it wasn't working properly, it would not be as big of a deal when it comes to affecting other components. The associated costs with this approach would be much cheaper as well. The downside is that it would be less accurate, as it would need to be retrained and updated periodically to improve the recommendation system.

INTERACTIONS

Approach One (Client):

As stated previously, the first approach of the proposed enhancement is the addition of a local client side recommendation system. From the proposed architecture, we can see a few new dependencies. The main dependency added here is a bidirectional dependency between the Content Database and the newly added Suggestion Algorithm. It should be noted that the Content Database here is a local repository of content that has been uploaded by the user. This dependency exists because the Suggestion Algorithm will need direct access to all of the user's uploaded content to make recommendations.

The other dependencies connected to the Suggestion Algorithm are one way dependencies on utils and on the User Info Server. These dependencies exist to allow the Suggestion Algorithm to make more informed decisions when trying to recommend content. Furthermore, the Suggestion Algorithm must be connected to the user's settings, to allow the user to configure their suggestions as they wish, or turn them off altogether. This results in the need for the Suggestion Algorithm to directly access the User Info Server.

Approach Two (Server):

The server side implementation of the content recommendation system is slightly more involved. For one, there are now two new components: the Content Aggregation Server, which acts as a database for all commercial content uploaded by users to Kodi, and the Suggestion Algorithm, as seen before, to create suggestions to users. In this implementation, the Suggestion Algorithm will not operate on the local Content Database, but instead it will formulate recommendations off of both the user's content consumption data as well as the large database of content it knows about, the Content Aggregation Server. This process adds some new dependencies to the Kodi architecture.

The main dependency in this new architecture are between the local Content Database and the server-side Content Aggregation Server. The reason this dependency exists is firstly to allow the Content Aggregation Server to include newly uploaded content from users into their Content Databases, and also to allow the Content Database to pull down content from the server when suggested and chosen.

The other dependencies in this server-side implementation of the content recommendation enhancement are the same as those found in the client-side implementation, and have been discussed previously.

TESTING PLANS

The first form of testing we will undergo is performance testing. We want to assess how a system performs under various conditions monitoring how our new feature affects the performance of the overall system. One tool we could use to do this is called loadrunner. Both for the client side and server side approach will conduct load testing to simulate real usage situations. We can use profiling tools to identify bottlenecks in our code and architecture, and we can monitor system resources such as CPU, memory, and disk I/O. An area of focus should be on requests from the backend on the server side approach as that is a failure point with higher probability.

Another important area of testing for a new feature is the user experience. We want to ensure that the system is intuitive and meets the users expectations. Focus testing is a great way to gather a diverse range of inputs on a new feature. Some common UX testing tools like UsabilityHub, and Lookback.io could also be used for our purposes.

When implementing server side approaches it is important to ensure that security is acceptable which is why we intend to incorporate security testing as well. We want to identify vulnerabilities and prevent unauthorized access. It's also important to ensure compliance with security standards and best practices. Using security tools such as Nessus and nmp, we can scan our system for vulnerabilities and conduct penetration testing.

Ideally we can automate some of these testing processes so that as the system evolves we can streamline testing and prevent shipping bugs. To prevent issues down the line we will document our tests, results, and anomalies for future reference. Lastly we will ensure throughout the process we keep the stakeholders involved being candied with the results of our testing. If we follow all these steps we can manage the risk involved with deploying a new feature.

POTENTIAL RISKS

For maintainability many potential risks could be present due to the complex nature of suggestion algorithms. As Kodi is an open source software if the creators of it did not provide ample documentation later fine tuning the model may be difficult, so it's important that the developers maintain well documented code. Performance wise suggestion algorithms can require large computational overhead potentially affecting the apps performance, overall with modern

devices this should not be a concern (Suggestion algorithms can be ran on old smartphones using instagram as an example, hence a modern computer should be able to run one with little performance concern). From a security perspective data privacy could be a potential concern to certain users. As suggestion models continuously evolve, every time a user is given a suggestion his data is used in the suggestion algorithm, which consequently could create minor adjustments to the algorithm as it would be self optimizing. It's important to note that overall this is a relatively low risk addition in context of the user's experience. If the model is given sub optimal suggestions the user can just choose to ignore them without significantly hindering the users experience on the app.

SEQUENCE DIAGRAMS

Diagram One: User Content Suggestions

Actors: Users

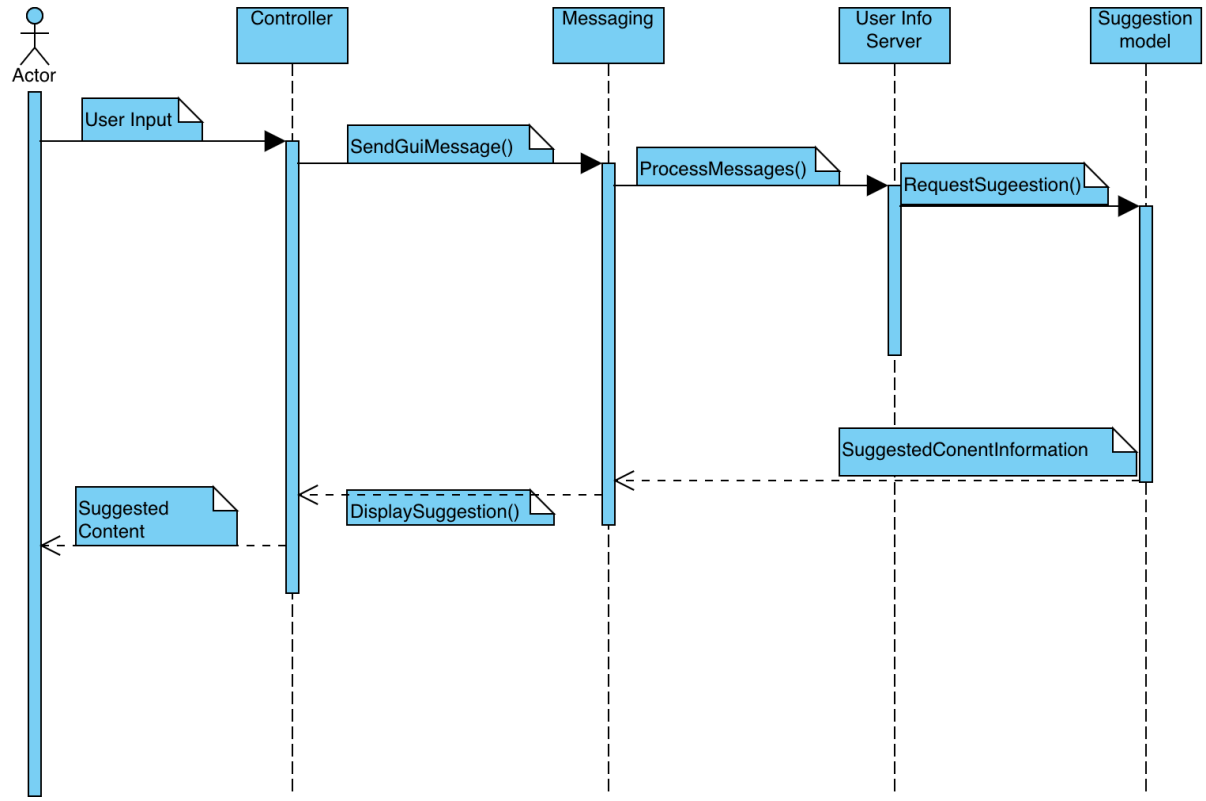

Figure 3: Sequence diagram for User Content Suggestions

This diagram depicts the suggested content being loaded for the user. It is initialized by the user's input, which in this case would be instantiated by the user clicking on a page where the content

suggestions would be visible. This action is sent from the controller using the SendGuiMessage() function, to the messenger, which is then processed and sent to the user info server. In this scenario the user info server would contain an input vector that would be used for the suggestion model. The input vector would be comprised of numerically quantified information of the content you have engaged with in the past. The vector is then sent to the suggestion model where a content suggestions array would be formed. This array is sent back to the messenger where it prepares the suggested content to be displayed to the user, then the re-formatted array is sent to the controller to be displayed.

Diagram Two: Updating User Content Information

Actors: Users

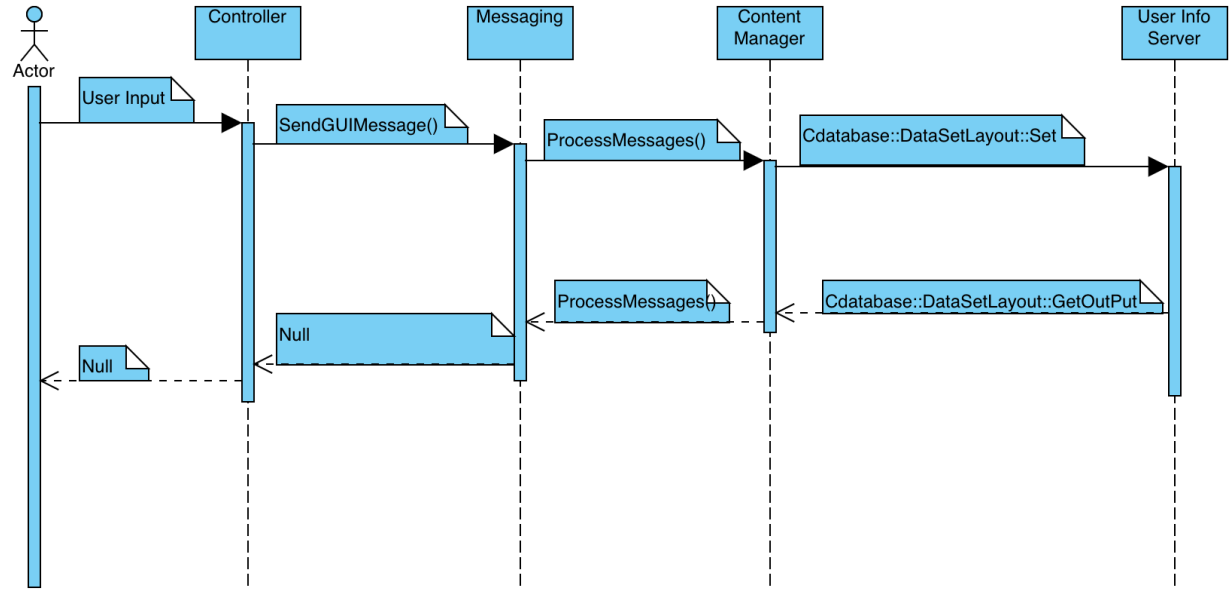

Figure 4: Sequence diagram for Updating User Content Information

This diagram depicts updating the user's input vector after he has consumed new content. As described above an input vector is used in the suggestions model. After a user consumes a new piece of media this vector should be updated. This process starts from the user's input, in this case when he stops consuming a piece of media. This triggers the SendGUIMessage() function from the controller, which then gets sent and processed in the messenger. The processed message is sent to the content manager which queries the user info server, looking to set a new value(s). The query output is sent back to the content manager, which sends the processed message back to the controller. Since the user is not informed of this happening no confirmation message is sent back to the controller or the user.

DATA DICTIONARY

Recommendation Algorithm (Client-side): A local algorithm in Kodi that generates content suggestions based on user interactions with various media types. It uses user-specific data to offer tailored recommendations.
Recommendation Algorithm (Server-side): A backend algorithm hosted on a server, which processes extensive user data to provide more comprehensive content suggestions. It continuously evolves based on user interactions.
Suggestion Algorithm Node: A new component in Kodi's architecture responsible for generating and sending content recommendations to users. It interacts with the ContentDB and other system nodes.
Content Aggregation Server: A server-side component that collects and manages data about available media content. It serves as a backend database for the server-side recommendation algorithm.
ContentDB Node: A component in Kodi that stores information about user interactions with media content. It is integral to both client-side and server-side recommendation systems for generating tailored suggestions.
User Info Server: Manages user-specific data, including preferences, watch history, and session-based information. It is crucial for personalizing content recommendations.
Utils Node: A module in Kodi responsible for various utility functions, including the display and management of recommendation suggestions within the user interface.
Testing Framework for Plugin Developers: A set of guidelines and tools provided to plugin developers to ensure seamless integration with Kodi's recommendation system.
Loadrunner: A performance testing tool used to simulate real usage situations and identify system bottlenecks, particularly in the context of the new recommendation features.
UsabilityHub & Lookback.io: User experience testing tools utilized to gather feedback on the new recommendation system and ensure it meets user expectations.
Nessus and Nmap: Security tools used for scanning the system for vulnerabilities and conducting penetration testing, especially important for server-side implementations.

NAMING CONVENTIONS

ClientSideRecSys: Refers to the client-side recommendation system in Kodi.
ServerSideRecSys: Denotes the server-side recommendation system implemented in Kodi.
SuggAlgoNode: Short for Suggestion Algorithm Node, a key component in the recommendation system.
ContentAggServer: Abbreviation for Content Aggregation Server, used in the server-side recommendation system.
ContentDB: Represents the Content Database node in Kodi, integral for storing user interaction data.

UserInfoServ: Abbreviation for User Information Server, managing user-specific data in Kodi.
UtilsNode: A shorthand for the Utilities Node, handling various utility functions in Kodi.
PluginTestFramework: Represents the Testing Framework provided to plugin developers for ensuring compatibility with the recommendation system.
PerfTest_Loadrunner: Refers to the use of Loadrunner for performance testing in Kodi.
UXTest_HubLookback: Represents the combination of UsabilityHub and Lookback.io tools for user experience testing.
SecTools_NessusNmap: A combined reference to Nessus and Nmap security tools used for system vulnerability scanning and penetration testing.

CONCLUSIONS

In conclusion, this report has thoroughly investigated the proposed enhancement of tailored content suggestions for the Kodi media platform. We explored two distinct implementation approaches: a client-side recommendation system and a server-side algorithm, each with its unique benefits and challenges. The client-side approach emphasizes a lightweight, modular system with easier integration and lower maintenance costs, albeit with potential limitations in accuracy and comprehensiveness. On the other hand, the server-side approach offers a more robust and evolving recommendation system, capable of leveraging extensive user data for more accurate suggestions, but at the cost of increased complexity and maintenance requirements.

Our analysis, including a detailed SAAM analysis, revealed the varying impacts of these enhancements on different stakeholders, including Kodi developers, independent plugin developers, and users. We emphasized the importance of balancing non-functional requirements such as performance, maintainability, testability, and security to meet the diverse needs and expectations of these stakeholders.

The proposed testing plans for both approaches aimed to address potential risks and ensure the seamless integration of the new features with Kodi's existing architecture. These plans covered essential aspects like performance, user experience, and security, highlighting the importance of a comprehensive testing strategy to maintain the high standards Kodi users are accustomed to.

Overall, the enhancement of tailored content suggestions presents a significant opportunity for Kodi to elevate its user experience, catering to the evolving needs of its diverse user base. While each approach has its trade-offs, careful consideration and implementation of these enhancements can lead to a more personalized, engaging, and user-friendly media platform. As Kodi continues to evolve, these enhancements will be crucial in maintaining its position as a leading media center software, adaptable to the changing digital landscape and user preferences.

LESSONS LEARNED

In the process of exploring and analyzing the enhancement of Kodi's architecture to include tailored content suggestions, several key lessons have emerged. These lessons not only reflect the specific challenges and triumphs of this project but also offer broader insights applicable to software development and system architecture enhancements in general.

1. **Balancing User Experience with System Complexity:** One of the primary lessons is the delicate balance between enhancing user experience and managing increased system complexity. While the introduction of tailored content suggestions undoubtedly improves user engagement, it also adds layers of complexity to the system. This complexity requires careful management to ensure system stability and maintainability.
2. **Adaptability in Architectural Design:** The exploration of client-side and server-side approaches underscored the importance of adaptability in architectural design. Systems must be designed with the flexibility to accommodate new features and technologies, as well as the foresight to anticipate future changes and challenges.
3. **Continuous Testing and Evaluation:** The project reinforced the need for continuous testing and evaluation throughout the development process. Regular testing, including performance, security, and user experience testing, is essential to identify and address issues early, ensuring the robustness and reliability of the system.

In conclusion, the journey of enhancing Kodi's architecture with tailored content suggestions has been rich with lessons that extend beyond this specific project. These insights are invaluable in guiding future projects that aim to enhance existing systems with new technologies and features, ensuring they are well-integrated, user-centric, and sustainable in the long term.

REFERENCES

"Add-on Development." Kodi Wiki, https://kodi.wiki/view/Add-on_development, Accessed 18 Nov. 2023.

Kodi Community Guidelines, https://kodi.wiki/view/Official:Kodi_Community_Guides. Accessed 18 Nov. 2023.

“Kodi Forum.” *Kodi Community Forum*, https://forum.kodi.tv/. Accessed 15 Nov. 2023.

Kodi Wiki. Official Kodi Wiki. (n.d.). https://kodi.wiki/view/Main_Page. Accessed 15 Nov. 2023.

Xbmc. “XBMC.” *GitHub*, https://github.com/xbmc/xbmc. Accessed 19 Nov. 2023.