CISC 322
A3

# SCUMMVM Architectural Enhancement


## Chamos:

Anes Numanovic 21an88@queensu.ca 20353836
Vasilije Petrovic 21vmp8@queensu.ca 20357533
Jacob Cruz 21jtc9@queensu.ca 20349571
Oliver Ren 21opr1@queensu.ca 20336697
Mattias Khan 22mhk@queensu.ca 20359237
Kevin Jiang 21kj31@queensu.ca 20353263

Table of Contents

## Abstract

This report proposes a new feature for ScummVM, “Cloud Save”. This feature serves to modernize the game emulator by enabling seamless save synchronization across platforms. This feature is implemented by the Cloud Save Manager, a new component that handles all cloud storage operations (e.g. uploading and downloading save data, managing local and cloud saves, etc.). This new feature will serve to enhance user convenience by letting them continue their playthrough from any platform. The approach to implement this feature was to introduce a dedicated cloud sync backend, which would handle synchronising modularly.

Following a more detailed overview, the report looks at two different ways this new feature could be implemented, including a SEI SAAM architectural analysis for both approaches. The impact of the feature on the architecture and subsystems at both the conceptual and concrete levels are also explored, along with the risks accompanying the changes. The report continues by diving into two use cases which help to show the beneficial value of the cloud save feature. Finally, a summary of the feature will be explored as well as a description of the lessons our team learned while deciding on a new feature for the ScummVM engine.

## Introduction

Many classic point and click adventure games have been preserved thanks to ScummVM. It is an open source software that interprets these game engines so they can run on modern systems. Ever since its launch in 2001, it has helped ensure many of these games do not get lost to time.

We dove deep into both the conceptual and concrete architecture in our previous two reports. This report introduces a new component: the Cloud Save Manager. One potential example is this: Imagine spending your time playing on your computer, and now you are going on a trip and will not have access to it. Cloud saves will allow you to pick up where you left off on your mobile device or Nintendo Switch. This module ensures seamless cloud saving.

We will first describe what this cloud save manager does and how it works. We lay out two potential approaches to implementing this: one with a centralized cloud service, and another which uses local-to-cloud sync via backends. We also show a conceptual architecture and discuss any new dependencies that were created by implementing this component. A SEI SAAM architectural analysis is also discussed. Next, we analyze how we might test this feature, as well any potential risks and limitations. Two use case scenarios regarding this new component are discussed. Finally, we summarize what we learned from the cloud save manager module.

## Overview of the New Feature

In today’s video game world, a major differentiator between ScummVM to other systems is the ability to synchronize game data across various devices, commonly through cloud storage. To level up ScummVM to modern standards, introducing a cloud save feature to the system would provide a significant enhancement to the user experience by allowing players to seamlessly access their saved games across multiple devices at any location. This feature would involve integrating cloud storage services to synchronize save files, ensuring that users can pick up their game exactly where they left off, regardless of the device they are using. Once enabled, save files would automatically upload and download as players save or load their progress. Overall, this change not only adds convenience but also aligns ScummVM with modern day gaming standards, making it more versatile and user-friendly.

## Approach 1: Centralized Cloud Service

For our first approach, our team proposes integrating a centralized cloud service to synchronize game data across devices. To go about implementing this feature, we would require a cloud service to be hosted and integrated directly into the system. In terms of cloud services, there is flexibility for server hosting; we can implement and host our own server or use existing cloud services such as AWS, GCP, or Microsoft Azure, among others. Once up and running, we would have to configure the system and cloud server to implement the cloud save functionality. On the ScummVM side, we would need to add/modify three items to make this work. Firstly, we would need to add cloud API integration to the common code layer, as well as implementation for a cloud manager module to handle authentication, upload, and download requests. Next, we would need to modify the GUI for cloud save management, by adding UI elements for logging in and out, browsing files, and manual synchronization. For this step we would also need to ensure that the GUI can communicate with the cloud manager in common code to trigger actions such as fetching or saving files. Lastly, we would have to modify save and load functions to automatically trigger uploads when a save is created or downloads when a game is loaded. On the server end of things, the following would need to be done: set up endpoints for upload/download operations (or custom API), set up authentication to handle user accounts securely, and add data access controls to enforce save size limits and to ensure data encryption during transit and storage for security.

Following this approach provides several advantages. For starters, the user experience will be very simplified as players can directly manage cloud saves from the interface. In addition, minimal additional infrastructure will be required if relying on a third-party cloud server hosting service, as many have default configurations which will have to be slightly modified to handle our task. However, although it might seem as if there is less work when hosting through a third-party, this could also provide certain disadvantages. For example, there could be unknown compatibility issues with reliance on third-party services, which would need extra time to render compatible, if possible. If not possible, we would have to switch hosting services altogether, or make our own. Of course, when it comes to using a third-party cloud server to host data, there are always security concerns to be had over storing save files in external services which would need to be addressed as well.

## Approach 2: Local-to-Cloud Sync via Backends

For our second approach, our team proposes implementing a dedicated cloud synchronization backend that interacts with a cloud server to sync game data across devices. This approach introduces an intermediary component, the cloud sync backend, which manages the synchronization of save files between the ScummVM system and the cloud server. Of course, same as in the previous approach, we would need to host a cloud server, either internally or through third-party services. On the ScummVM side, three key changes are required to make this approach work. Firstly, we would need to develop and integrate a cloud sync backend module. This backend component would handle queuing and managing the synchronization of save files. It would act as the bridge between the ScummVM engine, the common code, and the cloud server, ensuring modularity and isolating cloud-related operations. This backend would handle upload/download requests, manage sync statuses, and retry failed transfers automatically. Second, synchronization hooks need to be added to the save and load mechanisms. The ScummVM engine and common code layers would need to be updated with hooks that notify the cloud sync backend whenever a save file is created, updated, or accessed. This ensures that save files are synchronized in real time. Lastly, we would need to update the GUI for backend interaction. The GUI would need to provide

configuration options for the cloud sync backend. This includes enabling/disabling synchronization, account management, and manual synchronization actions. The GUI would also need to display these sync statuses or error notifications for user clarity. On the cloud server side, we would need to follow similar steps as in the first approach, with the following configurations needed: deploy a dedicated cloud server with APIs configured to handle upload/download/sync requests, set up authentication and access control through encryption and secure storage, and add synchronization logic and conflict resolution by maintaining metadata (i.e. timestamps and version histories) to ensure that latest save files are correctly synced, as well as adding conflict resolution strategies such as prioritizing newer files or asking users to choose which files to modify and how.

This approach provides several benefits as follows. Firstly, there is increased control and customization. By isolating cloud operations in the backend, we, the developers, gain full control over sync logic, error handling, and cloud-server interaction to allow for greater customization. Secondly, this design can handle growing user demand by scaling the backend and server independently, ensuring consistent performance as the system scales. Finally, this approach allows for easy integration of an offline support feature if needed in the future, as the backend can queue synchronization tasks for deferred uploads and downloads, to allow expansion for syncing even when offline. However, choosing to follow this approach will also pose disadvantages. The introduction of the cloud sync backend adds an extra layer to the architecture, increasing complexity and requiring significant development effort to implement and maintain. The addition of the backend may also introduce latency and additional processing overhead, especially for real-time synchronization tasks, paired with additional resource requirements required for deploying and managing the backend and cloud server.

*Comparison*

To highlight differences between each approach previously mentioned, the following is a table summarizing key features in a simpler visual format for ease of understanding.

| **Feature** | **Approach 1: Centralized Cloud Service** | **Approach 2: Local-to-Cloud Sync via Backends** |
|---|---|---|
| **Architecture Design** | Direct integration with a centralized cloud service | Introduction of a cloud sync backend to manage synchronization |
| **System Modifications** | Requires updates to the Common Code, GUI, and save/load functions for direct cloud integration | Adds a cloud sync backend, with synchronization hooks and GUI updates |
| **Cloud Server Requirements** | Requires hosting a cloud server with APIs for uploads, downloads and secure storage | Similar cloud server requirements, but with added sync logic and metadata management |
| **Offline Support** | Minimal support: will require many additional resources to implement | Strong support: sync backend task queues for deferred sync, little resources required to add |
| **User Experience** | Simplified user experience with | User-friendly GUI with room for |

| | | |
|---|---|---|
| | direct cloud save management in the GUI | adding an offline support feature to elevate player experience |
| **Complexity** | Less complex, fewer architectural changes required | Higher complexity, additional layer of backend architecture required |
| **Performance** | Dependent on cloud provider's latency & efficiency | Dependent on cloud provider as well + introduction of slight latency due to backend processing overhead |
| **Scalability** | Potentially limited by chosen provider's cloud service | Potentially limited by provider's cloud service, higher scalability however since backend and server can scale independently |
| **Development Effort** | Lower effort | Higher effort for elevated complexity to develop & maintain the sync backend |
| **Potential Risks** | Reliance on third-party cloud services can lead to compatibility or security concerns | Third-party reliance compatibility or security concerns, increased resource costs & complexity with added potential backend management challenges |

**Figure 1: Key Differences of Approach 1 vs. Approach 2**

## SAAM Analysis for ScummVM Cloud Save Enhancement

**Stakeholders:**

The primary stakeholders for the proposed cloud save enhancement are:

1. **Users**: Individuals using ScummVM to play games.
2. **Developers**: Contributors to the ScummVM open-source project.
3. **Investors/Project Sponsors**: Organizations or entities funding and supporting the project.

---

**Non-Functional Requirements (NFRs)**

**Users**:

- **Usability**: Ensure seamless access to saved games across devices and physical locations. Simple and descriptive UI informing users on cloud operations and errors, minimal user

input needed for account setup/management. Software also provides conflict resolution options (i.e. switching between local and cloud saves).

- **Reliability**: Ensure game progress is not lost or corrupted, network and upload/download failures are handled efficiently and descriptively. Ensure consistent uptime for cloud server(s).
- **Security**: Protects save data from unauthorized access or breaches. Data secured in transit and at rest. Authentication required for all operations on cloud server(s).
- **Performance**: Ensure synchronization latency of under 2 seconds under typical network conditions. Saving files has minimal impact on the performance of ScummVM during active gameplay. Ensure efficient handling of large save files with support for varying network speeds.

**Developers**:

- **Maintainability**: Ease of integrating and maintaining the new feature. Clear separation of cloud module(s) from the rest of ScummVM's architecture. Comprehensive documentation of new components and their dependencies.
- **Scalability**: Ability to handle increased user traffic without significant degradation in performance. Ensure independent scaling of synchronization is simple and has minimal to no impact on other system modules.

**Investors**:

- **Legal Compliance**: Meets data protection regulations (e.g., GDPR). Clear data retention and deletion policies for users save files. Regular audits of security measures to maintain compliance.
- **Cost-effectiveness**: Minimizes hosting and operational costs.
- **Brand Reputation**: Ensures a reliable and secure system to uphold trust. Provide users with transparency of security measures and data use as well as consistent consideration and implementation of user feedback.

---

**Evaluation of Approaches**

1. Centralized Cloud Service

**Users:**

- Usability: High, as the system simplifies file management through an integrated GUI.
- Reliability: Moderate; single-point-of-failure risk (server outages) could impact availability.
- Security: Moderate; data centralized on third-party servers may pose risks if the server is compromised.
- Performance: Dependent on server load and network conditions; could degrade during peak usage.

**Developers**:

- Maintainability: Moderate; requires updates to the ScummVM engine, GUI, and backends, plus integration with external APIs.
- Scalability: Limited by the capacity of the centralized server, requiring additional resources to scale.

**Investors**:

- Legal Compliance: Moderate; depends on third-party compliance with regulations.
- Cost-effectiveness: Potentially high operational costs if hosting is required. Security incidents could induce high costs to repair/restore stolen or compromised data.
- Brand Reputation: Could suffer during server outages or data breaches.

2. Local-to-Cloud Sync via Backends

**Users**:

- Usability: High, with modular management of synchronization and improved error handling.
- Reliability: High; localized issues affect only specific users rather than the entire system.
- Security: High; modularity reduces exposure, and synchronization logic enhances data integrity.
- Performance: High; distributed handling improves synchronization speed and reduces bottlenecks.

**Developers**:

- ❖ Maintainability: Moderate; modular backend introduces new dependencies but improves separation of concerns.
- ❖ Scalability: High; decoupling of backend and cloud server allows independent scaling.

**Investors**:

- ❖ Legal Compliance: High; modular approach aids in adhering to regulations.
- ❖ Cost-effectiveness: Better than the centralized model due to scalability and modular design.
- ❖ Brand Reputation: Enhanced by a robust and secure solution.

---

**Architectural Choice**

After evaluating both approaches, the **Local-to-Cloud Sync via Backends** is the superior option. It offers better scalability, security, and reliability while maintaining performance and usability. Although both approaches require significant development effort, the modular nature of the backend approach supports future extensibility and minimizes risks associated with single points of failure. This makes it a more sustainable and cost-effective choice, aligning with the needs of all stakeholders.

## Impact on Concrete Architecture

The implementation of this feature will require the addition of a new component within our architecture, as well as a change to each of the existing components. As a result of these changes, the concrete architecture itself also changed as seen below (Figure 2).

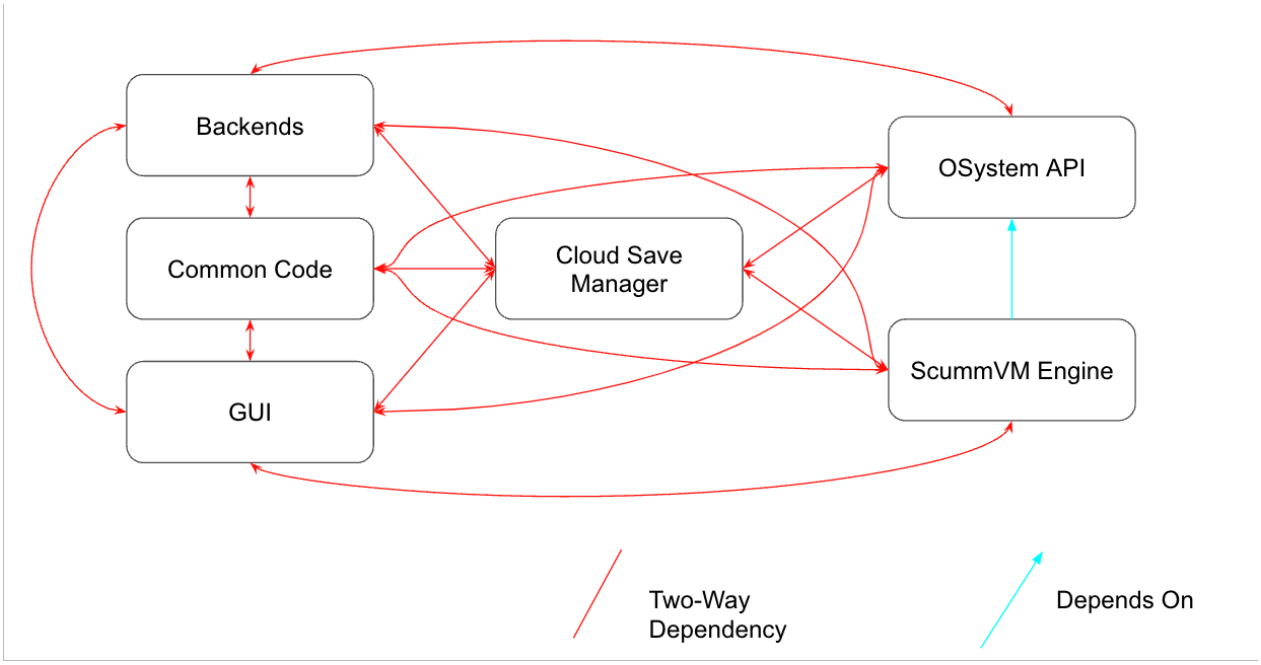



**Figure 2: Concrete Architecture**

*New Cloud Save Manager Component*

The cloud save manager is the new addition for this new feature. It's responsible for the cloud-based saving system that will save and load game progress remotely to be accessed by other devices / platforms. It works to abstract the cloud storage operations, such as uploading and downloading save files, and synchronizing progress across devices / platforms. Since this is a major feature, it will be getting its own component in our concrete architecture. It will have two-way dependencies to all the other components as well in order to ensure the save file isn't corrupted and interacts with the software properly, regardless of the platform being used. In terms of the interpreter architecture, the cloud save manager interacts with the other components by acting as an extra layer, handling and interpreting save file synchronization between local and cloud storage. In addition to this new component, a testing component should also be introduced to perform any integration or unit tests for the cloud save manager.

*ScummVM Engine*

The ScummVM engine module should be updated in order to properly interact with the cloud save manager. It needs to be able to perform save file interpretation, real-time updates, and game-state restoration. This component will have to be updated so that when a save file is created or modified, the cloud save manager is notified to synchronize the changes. To load a save state, the user will have to pull the file from the cloud save manager in order to interpret the game data accurately.

*GUI*

The GUI module will need to be changed to allow the user to interact with the cloud save state files rather than solely with the ScummVM engine. New code must be added to allow the user to customize the cloud save options, and resolve conflicts between local and cloud saves. Although the cloud save will be on by default, the GUI should allow the user to disable it if needed, as well as allowing them to organize their saves within the cloud and resolve conflicts. If the cloud save manager detects discrepancies between the local and cloud saves, the user should be prompted on what to do, e.g. delete the local and download the cloud save, vice versa, or delete both. Finally, the GUI should also interpret messages from the cloud save manager, prompting the user during each phase of the synchronization process (e.g. uploading save file, conflict detected, etc.).

*Backends*

Regardless of which platform is using the software, the cloud save manager should work with the backends to ensure that the save file is properly loaded no matter which platform is using it. The backends need to be updated to complement the cloud save manager, allowing it to execute actual network operations such as HTTPS requests to the cloud's storage. Additionally the backends would need to provide the platform-specific implementation for accessing the save file in question; this would allow the cloud save manager to interact with the local save directory.

*Common Code*

The common code is integral to each component of the architecture, and the cloud save manager is no different. This component must be updated in order to properly provide the shared utilities, data structures, and algorithms to the cloud save manager. The common code would provide the file I/O utilities to the cloud save manager so that it could interpret save file data efficiently. For example, it may use the common compression algorithm to compress the save file before it is sent to the cloud. Another needed update to common code would be to provide utilities for checksum generation or hashing; this would be integral for the cloud save manager which would use this to verify the save files during upload or download.

*OSystem API*

The OSystem API would need to integrate with the cloud save manager to allow it to work with the system's networking and file system efficiently. The OSystem API achieves this by abstracting these platform-specific operations in order for the other components to more easily interact with the system. The cloud save manager would use this component in order to interpret synchronization commands as well as delegate the platform-dependent network requests such as HTTP POST in order to save a file. This component would also need to be updated so that reading and writing save files is consistent across different platforms.

## Testing

Before we can integrate our new proposed feature of cloud saving into our existing ScummVM system, we will first need to perform unit testing on it. This entails testing the Cloud Save Manager component by itself separate from any other portions of our system. For our unit test, we will test if the Cloud Save Manager component can carry out its required functionality. This includes important tasks such as uploading save data and retrieving save data. On the unit level, we will also test for certain non-functional requirements. The performance of the cloud save manager will be tested. The security of the cloud save manager will be tested, including things such as its encryption quality, its integrity, and its authentication system. How reliably the cloud save manager can transmit data without corruption will also be tested.

After that, we will perform integration testing on this system. This involves testing if our cloud manager module works when integrated with our existing ScummVM system. To test this, we will examine the ways in which the cloud save manager module interacts with existing modules of the ScummVM architecture. This will be highly thorough as the Cloud Save Manager has numerous interactions with all major components of ScummVM.

After integration testing is complete, the cloud saving feature will be integrated to ScummVM. After this integration is complete, regression testing will be performed. The purpose of these tests are to verify that integrating this new feature does not cause our system to malfunction. The regression testing will involve 3 steps. First we will test that the changed system still carries out all of its functionality correctly. Then we will test if integrating this feature has reintroduced old errors. Finally we will test if the new system is still effective in an operational production environment.

The testing process will involve the use of manual testing, automated testing, and open beta testing. Automated testing will be used to add consistency and reduce human error and labour. Open beta testing allows for us to take advantage of ScummVM's open source nature, and allow volunteers to help test ScummVM in a production environment.

## Potential Risks and Limitations

*Scalability*

As a low entry barrier open source project, ScummVM's user base will continue to grow larger in the future. This introduces the possible risk of insufficient scalability. To account for an increasing user count, ScummVM's cloud server must be highly scalable. If this is not the case, this growth can hurt the application in numerous possible ways. With more users using ScummVM, the amount of cloud requests will also increase. The cloud may not be able to process higher request loads, causing long response times which may be inconvenient for users. Moreover, more users also mean that more users may be connected to the ScummVM cloud at once. the cloud may not be able to handle the increased amount of connections

being made. This may lead to servers crashing and becoming unavailable, leading to further inconvenience for users.

*Security*

Involving a server can introduce numerous security risks to the application. Since this service will require account authorization, possibly including information such as email addresses and passwords, the servers may become a target for malicious hackers who aim to steal this important security information. One common avenue for this may be through social engineering attacks. These involve manipulating users into sharing security related information, through techniques such as phishing. Hackers can also directly target ScummVM servers to harm the application. An example of this include denial of service attacks, which involves a malicious actor overwhelming a server with artificial traffic in order to deny the server's service. These security issues have the potential to put the application and its users at risk.

*Maintenance Fees*

This project may be limited by the development team's ability to pay for server maintenance fees. Maintaining a cloud powerful enough to support ScummVM's large user base will be expensive. This becomes a problem as it conflicts with ScummVM's open source development methodology. Many unpaid volunteer developers contribute to ScummVM's development and accessibility is greatly valued. Because of this, ScummVM is a freely downloadable software and the development team does not profit directly from ScummVM users. Because of this, ScummVM may lack a revenue source that can pay for the server maintenance fees.

## Use Cases

Diagram for loading save:

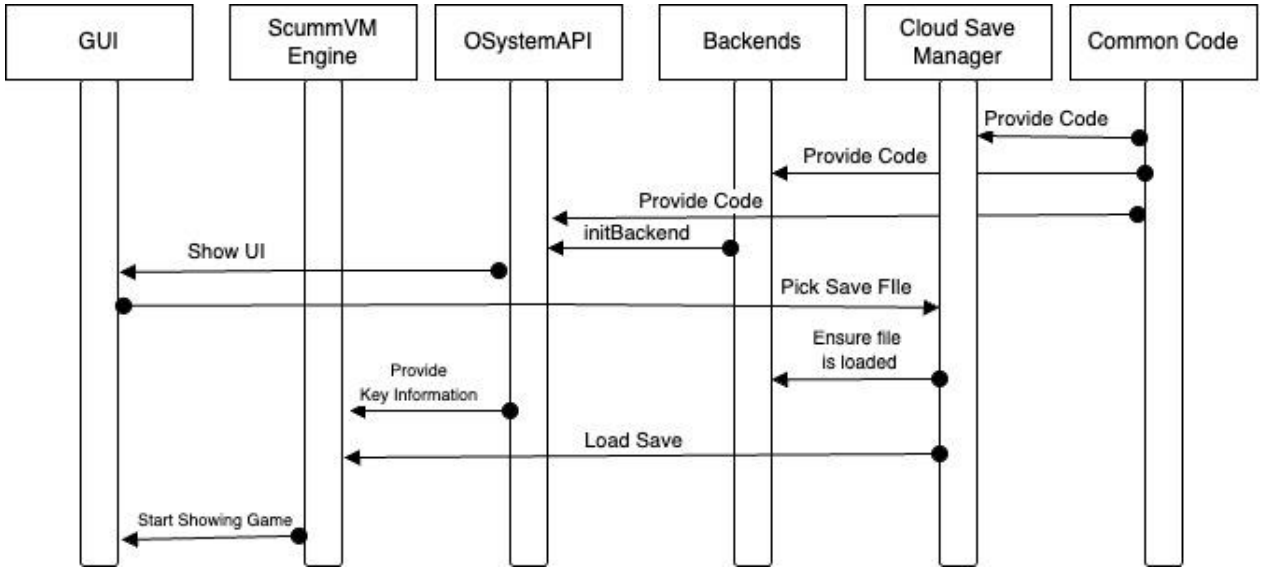
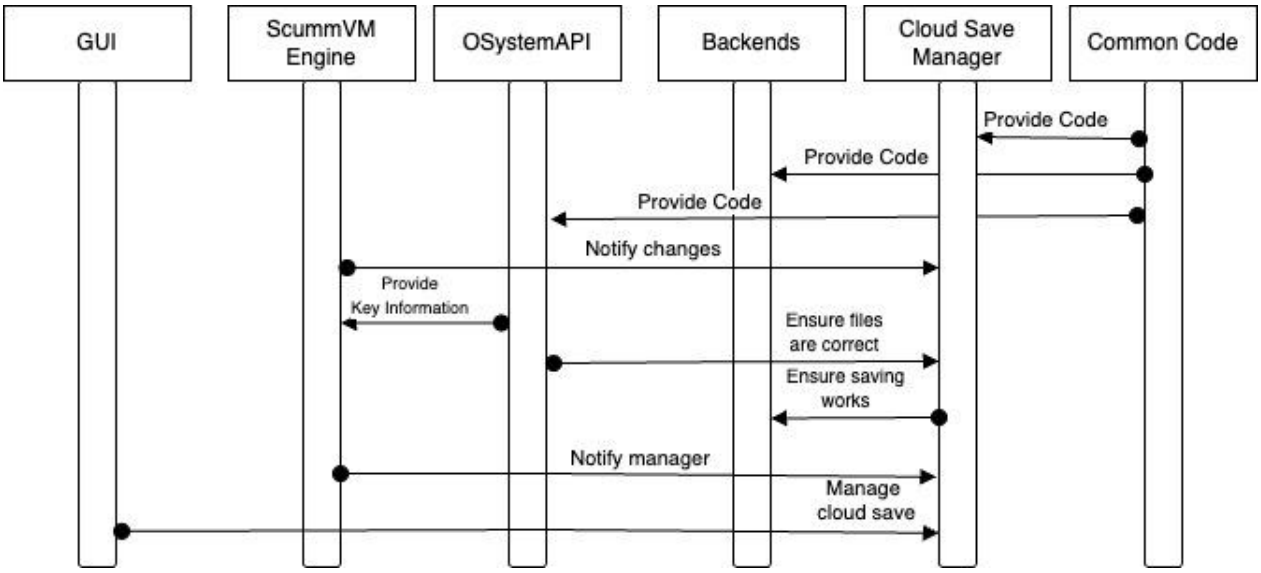



Diagram for saving to cloud save:




The first use case shows what happens when the user selects a save file from the cloud save manager to play. First, common code is provided to the various components, and the backend and UI are initialized. Next, the user picks a save file they wish to load from the cloud save manager. The backends ensure the save file is loaded correctly. Finally, the ScummVM engine loads the cloud save, and then the game is displayed on the GUI.

The next use case shows the user saving to the cloud save manager. The common code provides important code to the components again. The ScummVM engine notifies the cloud save manager of any changes in the game state. The OSystemAPI ensures the system's files are correct. The backend also ensures saving works. The ScummVM also notifies the cloud save manager of any changes in the game state. Finally, the GUI allows users to interact with the cloud save manager, including saving and managing options.

## Lessons Learned

Creating the report on ScummVM's cloud save feature enhancement provided valuable insights into architectural decision-making and software development challenges. We realised the importance of evaluating trade-offs between centralized and modular solutions, emphasizing scalability, security, and maintainability. We gained a deeper understanding of integrating new components into legacy systems while balancing user needs and technical constraints. The process highlighted the need for comprehensive testing strategies, including unit, integration, and regression testing, to ensure the robustness of enhancements. Addressing potential risks, such as server scalability and security vulnerabilities, reinforced the critical role of proactive risk management. We also learned to collaborate efficiently with one another, communicating project needs, sharing ideas, and problem solving cohesively as a group. Overall, this project taught us the importance of structured analysis, stakeholder engagement, and iterative development in creating sustainable and scalable software solutions.

## Conclusion

In conclusion, we plan to enhance ScummVM by adding to it a cloud saving feature that allows for game data synchronization across multiple devices. We have proposed two approaches for implementing this feature. One approach involves using a centralized cloud service. The second approach involves implementing a cloud synchronization backend that interacts with a cloud server. We then performed an SEI SAAM architectural analysis on this new feature. The Users, Developers, and Sponsors were identified as the project's primary stakeholders. We then listed the Non-Functional Requirements of each stakeholder group and how each approach would satisfy these NFRs. This analysis concluded that our second approach involving the backend would fulfil these requirements most effectively, and would therefore be the approach we would use. Adding this feature necessitates implementing a new Cloud Save Manager component to the architecture. This component is responsible for handling the application's interactions with the cloud, and has 2-way dependencies with the Engine, GUI, Common Code, Backends, and Osystem API components. To account for this, these aforementioned components must also be changed to properly interact with the Cloud Save Manager. We updated our concrete architecture to include the Cloud Save Manager and the new interactions that come with it. To test this new feature, we will use a combination of unit testing, integration testing, and regression testing. We then examined the limitations and risks of this enhancement, which included scalability, security, and maintenance fees. Finally, we illustrated how this updated system would handle the use cases of loading a save and saving to the cloud.

## Data Dictionary

### Naming Conventions

- **AWS**: Amazon Web Services
- **GCP**: Google Cloud Platform
- **NFR**: Non-Functional Requirement

## References