Bilkent University
Department of Computer Engineering

Senior Design Project
T2527
CollabHub

Analysis and Requirement Report

Tuna Göksal | 22203827
Yiğit Özhan | 22201973
İbrahim Çaycı | 22103515
Moin Khan | 22101287
Ömer Edip Aras | 22203238

Supervisor : Ayşegül Dündar Boral
Course Instructors : Mert Bıçakçı, İlker Burak Kurt

19/12/2025

This report is submitted to the Department of Computer Engineering of Bilkent University in partial fulfilment of the requirements of the Senior Design Project course CS491/2.

Contents

1 Introduction **3**
1.1 Purpose of the system 3
1.2 Design goals 3
1.3 Definitions, acronyms, and abbreviations 4
1.4 Overview 4
2 Current Software Architecture **5**
3 Proposed Software Architecture **6**
3.1 Overview **6**
3.2 Subsystem Decomposition 6
3.3 Hardware/Software Mapping 9
3.4 Persistent Data Management 10
3.5 Access Control and Security 10
3.6 System Models **12**
3.6.1 Use-Case Model **12**
3.6.2 Object and Class Model **13**
3.6.2.1 Class Diagram 13
3.6.2.2 Activity Diagrams 14
3.6.2.3 Sequence Diagrams 16
4 Subsystem Services **17**
4.1 GUI Layer 17
4.2 Business Logic Layer 19
4.3 Data Layer 23
5 Test Cases **24**
5.1 Functional Test Cases 24
5.2 Non-functional Test Cases 29
6 Consideration of Various Factors in Engineering Design 33
6.1 Constraints 33
6.1.1 Implementation Constraints 33
6.1.2 Economic Constraints 33
6.1.3 Ethical Constraints 33
6.1.4 Social Constraints 34
6.2 Standards 34
7 Teamwork Details **34**
7.1 Contributing and Functioning Effectively on the Team 34
7.2 Helping create a collaborative and inclusive environment 35
7.3 Taking lead role and sharing leadership on the team 36
8 Glossary **36**

9 References 37

1 Introduction

1.1 Purpose of the system

The purpose of the *CollabHub* system is to provide a version-control and collaboration platform specifically designed for architectural design projects using Autodesk Revit (.rvt) files. Traditional version control systems, such as Git, are designed for text-based files and are not suitable for managing large binary files, such as architectural models [1]. CollabHub aims to solve this problem by enabling architects to track changes, manage multiple versions of their models, and collaborate more effectively with team members. The system focuses on detecting changes within RVT files and synchronizing them across users while maintaining version history.

1.2 Design goals

Usability

1. The user interface will be integrated directly into the existing workflow to ensure that architects can easily perform version control tasks without additional training.
2. Core actions such as pushing and pulling updates will be clearly presented and simple to execute.
3. The system will provide clear and understandable feedback messages indicating the success or failure of an operation.

Reliability

1. A version will only be stored if all associated data has been fully and correctly received by the backend.
2. The system will report common errors such as invalid data or network failures to the user.
3. Incomplete or corrupted differences will not be stored or applied.

Performance

1. The system will conceptually improve efficiency by transmitting only differences between versions rather than entire files.
2. The client is expected to behave with responsiveness typical of modern desktop applications, without strict numerical performance targets.

Supportability

1. The system will maintain a modular internal structure to make future enhancements and maintenance easier.
2. Basic logging of push, pull, and error events will be implemented to support debugging.

3. Developer documentation appropriate for a university-level project will be provided.

Scalability

1. The backend will be structured to logically support multiple projects, even if testing is limited to a smaller scope.
2. The version storage structure will be designed so that it can be extended in future phases without major redesign.

1.3 Definitions, acronyms, and abbreviations

The following terms are used throughout this report:

- **API Gateway:** The server-side routing component that manages request routing and JWT validation.
- **Commit:** A recorded version entry in the system that stores information about a change, including associated metadata such as user and timestamp.
- **Commit History:** The chronological record of previous commits displayed by the system together with metadata such as timestamps and authors.
- **Diff / Differences:** The computed change information between two commits or snapshots used for comparison and synchronization.
- **Entity:** A defined data structure stored within the system, including user records and snapshot-related data.
- **FastAPI:** The framework used to implement the backend service of the project.
- **Meta Database:** The PostgreSQL relational database that stores non-geometric project metadata such as commit logs, user information, and branch pointers.
- **Repository:** The data access interface that executes database queries, manages storage I/O, and abstracts database mechanisms from the business logic layer.
- **RVT Version Control Engine:** The component responsible for tracking file lineage within the system architecture.
- **Snapshot:** The stored representation of architectural element or model state used for comparison, synchronization, and version reconstruction.
- **Synchronization:** The process of updating the local model so that it matches the latest shared project state.
- **Version Control:** The set of system operations used to track, compare, and manage project changes over time through commits, snapshots, push, and pull operations.

1.4 Overview

Architectural design software has significantly improved with advances in computer technology. Tools such as Autodesk Revit allow architects to create detailed three-dimensional models of buildings. However, collaboration on these models remains difficult because the architectural files are large and use proprietary formats. This makes it hard to use traditional version control systems that are commonly used in software development.

In many cases, architects share updated files via cloud storage or network drives, which can lead to overwritten work, missing versions, or confusion about which file is the latest. Existing commercial collaboration tools also have limitations, including high costs and limited accessibility for students or smaller teams.

To address these issues, this project introduces **CollabHub**, a version control system designed specifically for architectural models. The system integrates with the Revit environment and allows users to push changes, pull updates, and compare different versions of a model. By focusing on the structure of RVT files and transmitting only the differences between versions, CollabHub aims to provide a more efficient and accessible solution for architectural collaboration.

2 Current Software Architecture

Architectural design collaboration is currently handled through a combination of file-based workflows and proprietary coordination platforms. In most practice environments, architects work on local copies of large RVT files and exchange updated versions via shared network drives, cloud storage services, or email. Version tracking in this process is informal and typically relies on manual file naming conventions or external documentation. As the number of collaborators and project complexity increase, this approach often leads to overwritten changes, loss of intermediate design states, and uncertainty about who made a particular modification and when.

Several commercial tools attempt to improve collaboration by offering centralized, cloud-based environments. Autodesk BIM Collaborate enables teams to publish and review models using Autodesk's cloud infrastructure and supports workflows such as design coordination and clash detection [2]. While these features improve accessibility and coordination across teams, they operate primarily at the level of complete model versions and require commercial licensing. During informal discussions conducted with architecture students, it was noted that BIM Collaborate is not included in standard student licensing plans, which limits its accessibility in academic settings.

Similarly, platforms such as 3D Repo provide cloud-based model visualization, issue tracking, and comparison between uploaded revisions[3]. These tools are effective for design review and coordination, but treat architectural models as review artifacts rather than actively versioned design assets. They do not manage the internal structure of proprietary RVT files, nor do they support controlled merging of concurrent changes directly within the authoring environment. Feedback gathered from architecture students also indicated that subscription costs for tools such as 3D Repo and similar BIM coordination platforms are considered prohibitively expensive for regular student use.

Generic version control systems such as Git are not suitable for architectural workflows. RVT files are large proprietary binary files, making textual differencing impractical and storage inefficient. These systems lack awareness of architectural semantics and cannot provide visual feedback or safe conflict resolution for geometric and parametric changes.

As a result, existing solutions focus on file sharing, cloud-based coordination, or model review, often with significant cost barriers and without providing structured version control tailored to architectural design files. In contrast, CollabHub is a free system developed in an academic context, prioritizing accessibility alongside technical capability. By avoiding licensing barriers and focusing on RVT-specific version tracking, branching, and controlled merging, CollabHub aims to make advanced collaboration mechanisms available to a broader range of users, particularly students, small teams, and research-oriented environments.

3 Proposed Software Architecture

3.1 Overview

CollabHub is designed as a specialized version control platform engineered exclusively for the architectural domain, with strict support for the .rvt file format. The system allows multiple architects to collaborate within a shared project space, providing features similar to GitHub or Bitbucket but tailored for the visual nature of architectural design. The main objective is to prevent the limitations imposed by generic frameworks by utilizing a core engine optimized to track changes specifically within .rvt files.

The CollabHub system utilizes a client-server architecture designed so as to handle the high bandwidth and processing requirements of large 3D architectural files.

- **Client Side:** The client application serves as the interface for the architect. It is built as a Graphical User Interface (GUI) provided as a plugin to the RVT development environment, combining development and version control environments. It includes a Desktop GUI for history management, an RVTFileWatcher background service, and a Local RVTViewer.
- **Server Side:** The backend is designed for scalability and efficiency. It includes an API Gateway for request routing, an RVT Version Control Engine to track file lineage, and handle heavy RVT data efficiently.

3.2 Subsystem Decomposition

The CollabHub system architecture is decomposed into three distinct layers to manage the processing and bandwidth requirements of 3D architectural files. This structure separates the user interface, core processing logic, and persistent storage mechanisms.

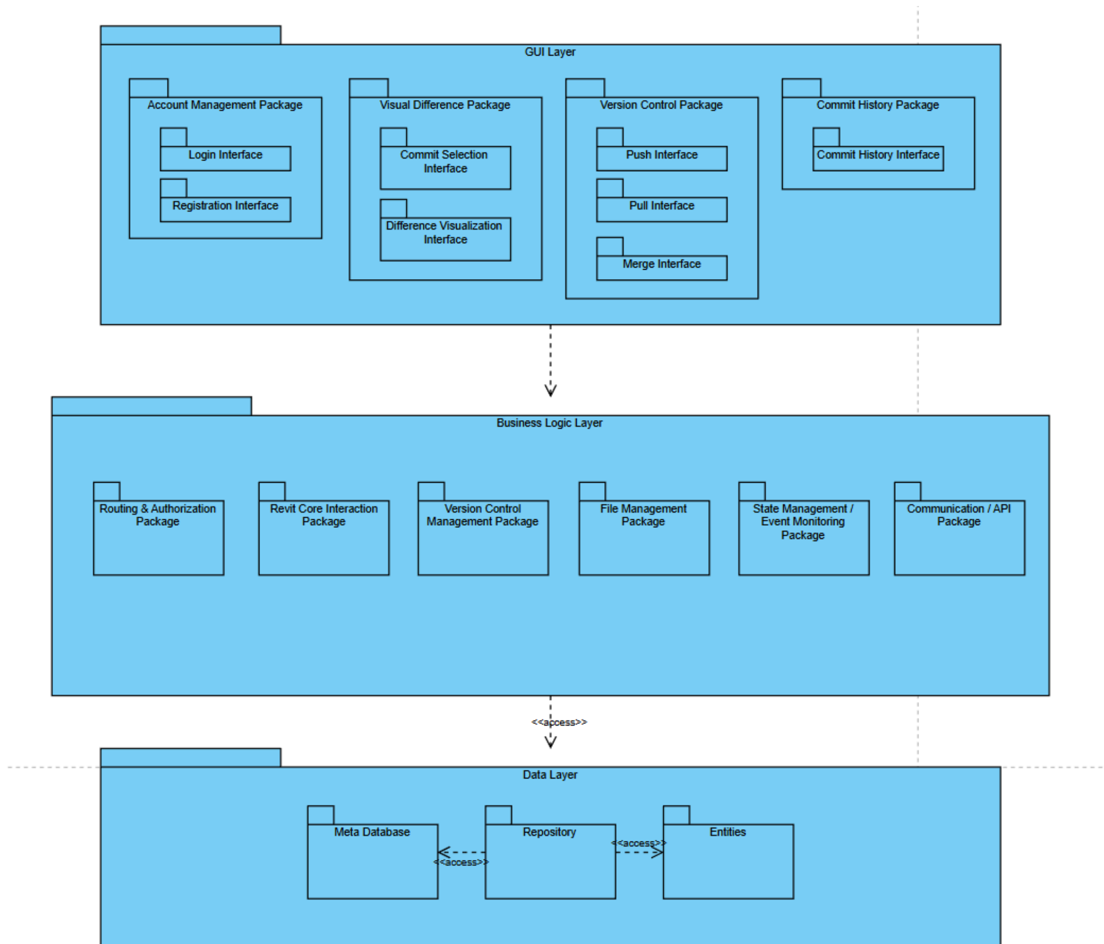
The GUI Layer functions as the client interface, operating as a plugin within the Autodesk Revit environment. It accesses the Business Logic Layer to execute user commands. The Account

Management Package handles user authentication through login and registration interfaces . The Visual Difference Package renders color-coded analysis of geometric changes, such as additions, modifications, and deletions, directly within the Revit 3D viewport, and includes commit selection interfaces . The Version Control Package provides the push, pull, and merge interfaces, allowing users to synchronize local changes with the central server and resolve conflicts . The Commit History Package displays a chronological list of previous commits, branches, and author metadata.

The Business Logic Layer distributes core processing tasks between the client-side plugin and the server-side backend . The Routing & Authorization Package functions as the API Gateway on the server, managing request routing and JWT validation. The Revit Core Interaction Package utilizes an extraction component to query the active Revit model's database, extracting geometric and parameter data into a structured format. The Version Control Management Package contains the engine that tracks file lineage, computes differences between base and target snapshots, and detects concurrent modification conflicts. The File Management Package handles operations specific to the proprietary .rvt file format. The State Management / Event Monitoring Package operates a background service that monitors local file saves to prompt users for commits. The Communication / API Package manages network transmission, ensuring that only differential data is sent to the server instead of the entire project file.

The Data Layer manages the persistent storage and retrieval of all system information. The Meta Database is a PostgreSQL relational database that stores non-geometric project metadata, including commit logs, user information, and branch pointers. The Repository package provides the data access interface that executes database queries, manages storage I/O, and abstracts the underlying database mechanisms from the Business Logic Layer. The Entities package represents the defined data structures stored within the system, including AES-256 encrypted architectural assets, JSON snapshots, and structured user records.

.


GUI Layer
Account Management Package
Login Interface
Registration Interface
Visual Difference Package
Commit Selection Interface
Difference Visualization Interface
Version Control Package
Push Interface
Pull Interface
Merge Interface
Commit History Package
Commit History Interface
Business Logic Layer
Routing & Authorization Package
Revit Core Interaction Package
Version Control Management Package
File Management Package
State Management / Event Monitoring Package
Communication / API Package
<<access>>
Data Layer
Meta Database
Repository
Entities
<<access>>
<<access>>

3.3 Hardware/Software Mapping

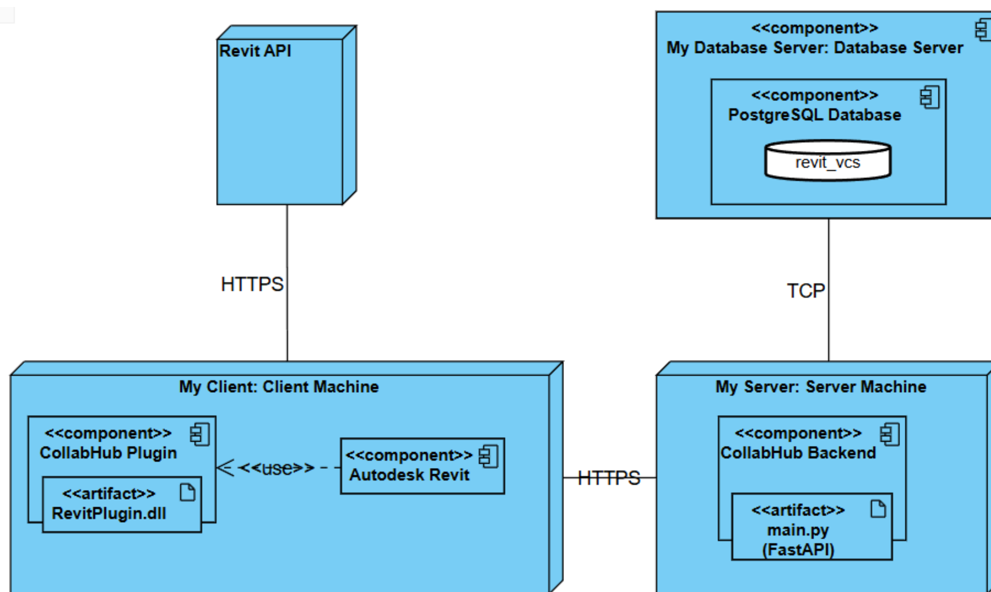


Figure: Deployment Diagram

The *CollabHub* system follows a client–server architecture. The system consists of three main parts: the client machine, where Autodesk Revit and the *CollabHub* plugin run; the backend server, which handles requests and version-control logic; and the database server, which stores project data.

As shown in the deployment diagram, the client side runs on the user's computer. Architects interact with the system through a plugin integrated into Autodesk Revit. The plugin is implemented as a dynamic library and communicates directly with the Revit API. With this plugin, users can push their changes to the server, pull updates from other collaborators, and view the version history. The plugin sends requests to the backend server using HTTPS.

The server-side runs the system's main application logic. The backend is implemented in Python using the FastAPI framework. The server receives requests from the Revit plugin, processes them, and manages operations such as version tracking, snapshot storage, and retrieval of project history. This component acts as the central coordinator between the client plugin and the database.

The database server is responsible for storing persistent data. *CollabHub* uses a PostgreSQL database to store metadata such as commits, project information, and version snapshots. The backend server communicates with the database via the TCP/SQL protocol to read and write data as needed.

From a deployment perspective, the workflow is straightforward. The *CollabHub* plugin communicates with the backend server through HTTPS, and the backend server communicates with the PostgreSQL database using SQL queries. The plugin also interacts locally with the Revit API to extract model information and apply architectural model updates.

Regarding hardware requirements, the client machine simply needs to be capable of running Autodesk Revit, since the plugin runs inside the Revit environment. Therefore, any computer that meets the normal Revit system requirements should be sufficient. The server machine only needs enough resources to run the FastAPI backend and the PostgreSQL database, which can run on a standard server or cloud instance.

3.4 Persistent Data Management

The persistent data of *CollabHub* is stored in a PostgreSQL database, which maintains the information required for version control, user management, and project history. The database stores metadata related to users, projects, commits, and element snapshots that represent the state of architectural models at different points in time. These records allow the system to track changes and maintain a consistent history of modifications made to Revit models.

Data is created or modified only when users perform specific actions through the CollabHub plugin, such as creating a project, pushing a new version, or retrieving project history. When a user pushes changes from the Revit plugin, the backend extracts the relevant architectural element data and stores the associated metadata in the database. Each commit entry includes information such as the user who made the change, the timestamp of the operation, and references to the stored model data. This structure allows the system to reconstruct previous versions of the model and maintain an organized version history.

The backend server manages all interactions with the database through controlled API operations. The CollabHub plugin does not communicate directly with the database; instead, all data access is handled by the FastAPI backend. This ensures that data operations remain consistent and prevents unauthorized manipulation of stored information.

In addition to commit and project information, the system maintains user-related data required for authentication and project collaboration. When users register or log in to the system, their credentials and account information are stored securely in the database. These records are used by the backend to verify user identity and associate project actions with specific users.

Since architectural models may evolve frequently during the design process, the database structure is designed to support continuous updates. Each new commit creates a new entry in the system without overwriting previous records. This approach ensures that historical versions remain available and that project evolution can be traced over time.

Overall, the persistent data management approach in CollabHub ensures that project data, version history, and user information are stored reliably while supporting collaborative workflows between multiple architects working on the same model.

3.5 Access Control and Security

Since *CollabHub* manages architectural design files and project history, it is important to ensure that only authorized users can access or modify project data. The system includes several

access control and security mechanisms to protect project files, user information, and version history stored on the server.

First, CollabHub requires users to authenticate before interacting with the system. The backend server handles authentication using JSON Web Tokens (JWT). When a user logs in through the Revit plugin, their credentials are verified by the backend. If the authentication is successful, the server generates a JWT token which is then used for subsequent requests. This token-based approach ensures that only authenticated users can perform operations such as pushing changes, pulling updates, or viewing project history.

In addition to authentication, the system associates each action with a specific user account. Every commit or change uploaded to the server is linked to the user who created it. This provides accountability and allows the system to maintain a clear history of who made specific modifications to a project.

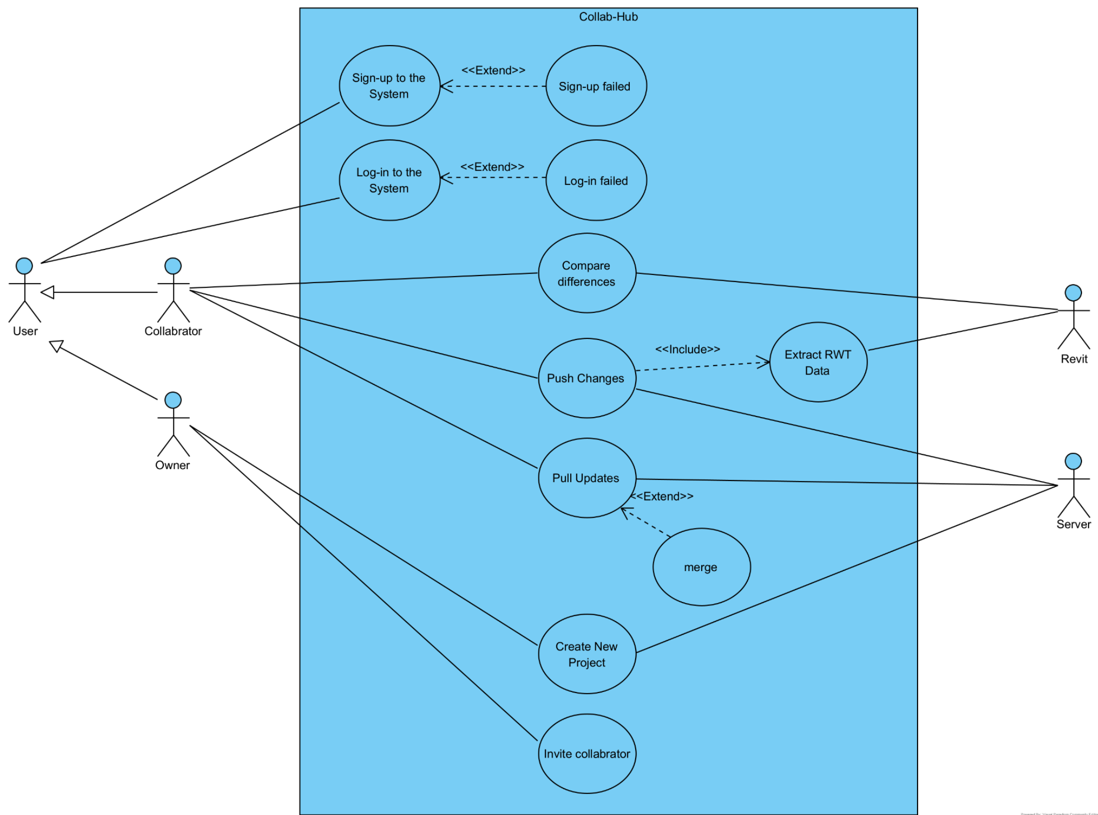
Communication between the *CollabHub* plugin and the backend server is performed through HTTP requests secured with HTTPS. Using HTTPS prevents unauthorized parties from intercepting sensitive data such as authentication tokens or project metadata during transmission.

Data stored on the server is also protected through secure database access. The backend communicates with the PostgreSQL database using controlled queries, ensuring that only the backend application can read or modify stored project data. This prevents direct external access to the database and reduces the risk of unauthorized data manipulation.

Together, these mechanisms provide a basic but effective security model for the *CollabHub* system. Authentication, secure communication, and controlled database access help ensure that architectural project data remains protected while still allowing multiple collaborators to work on the same project.

3.6 System Models

3.6.1 Use-Case Model



3.6.2 Object and Class Model

3.6.2.1 Class Diagram

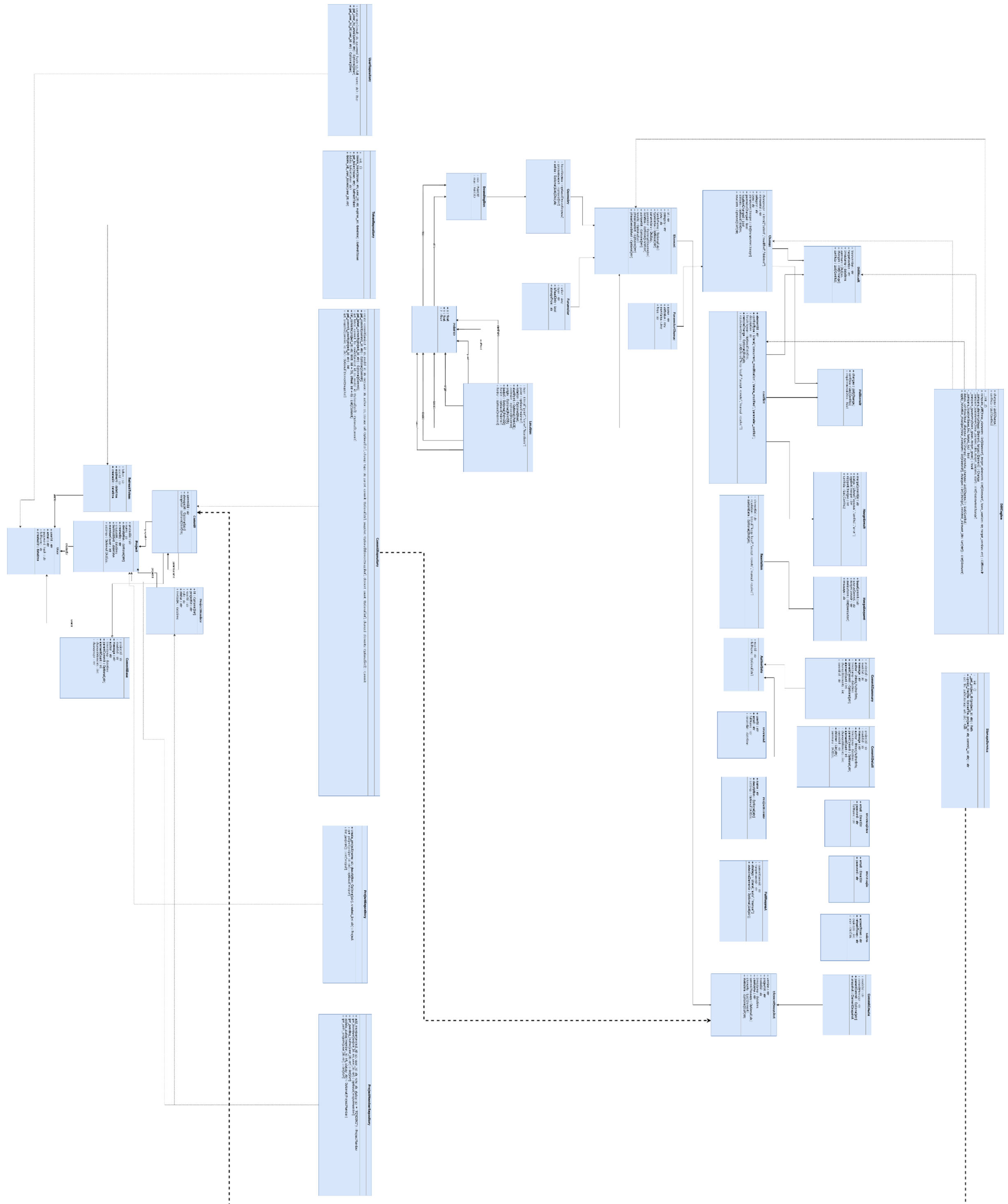
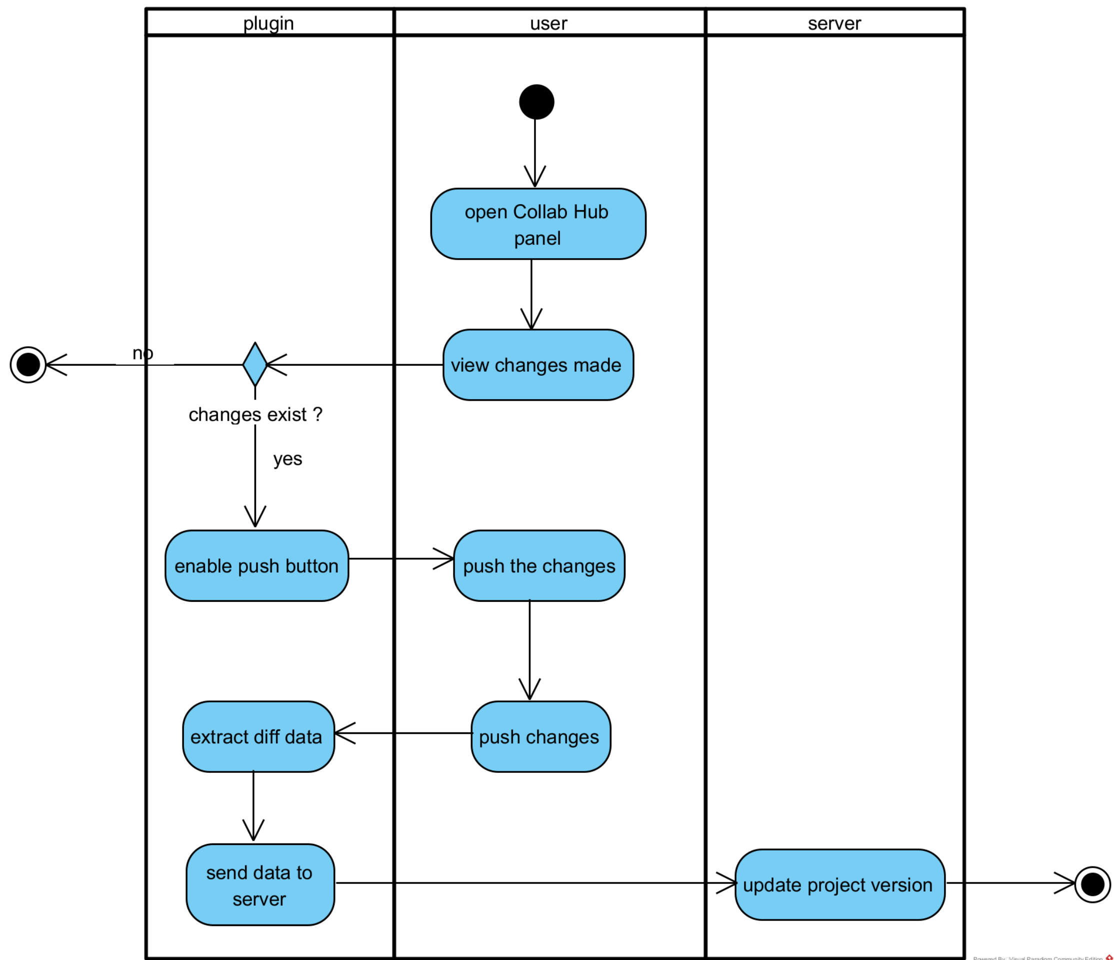
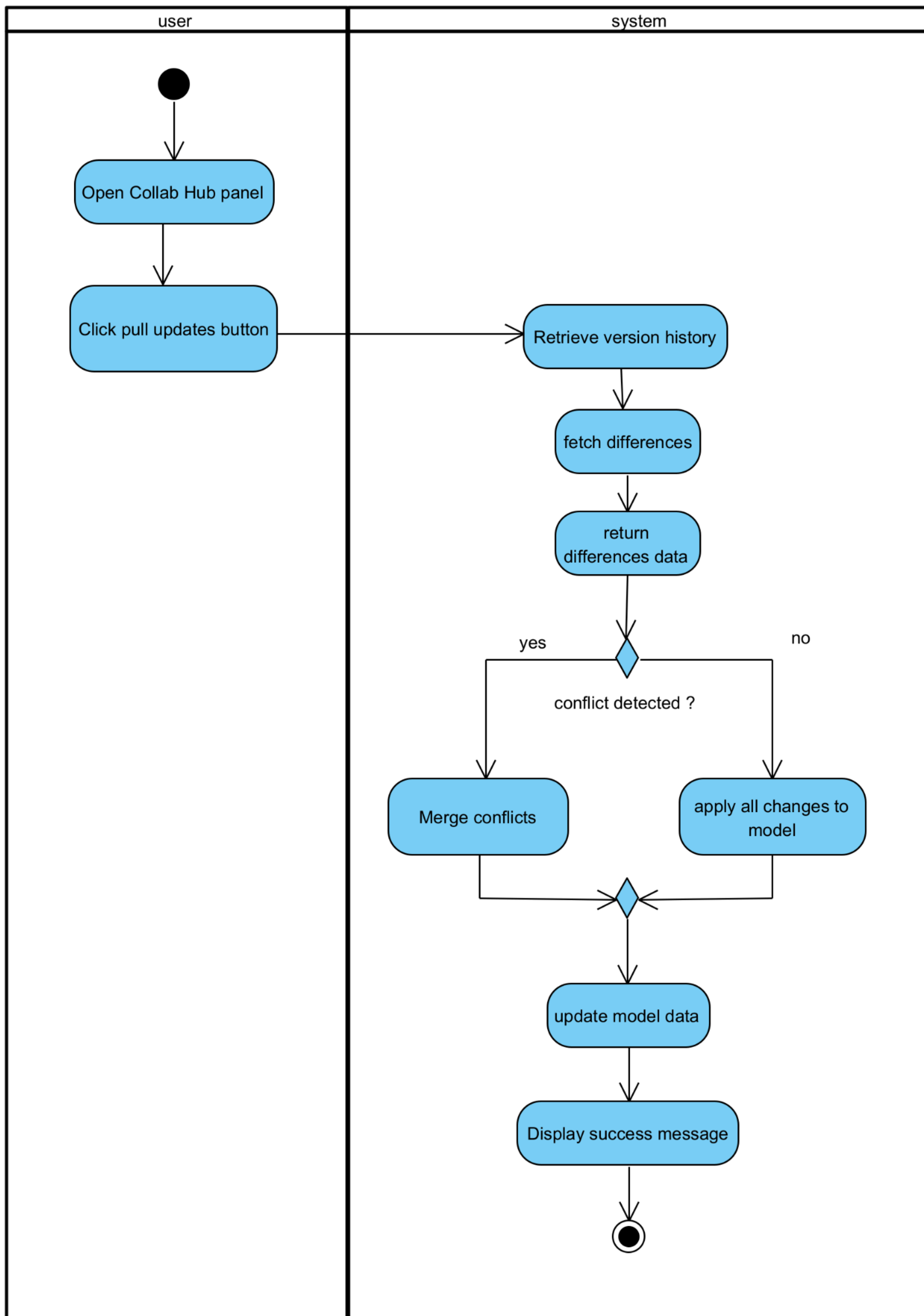


Figure : Class & Object Diagram

3.6.2.2 Activity Diagrams



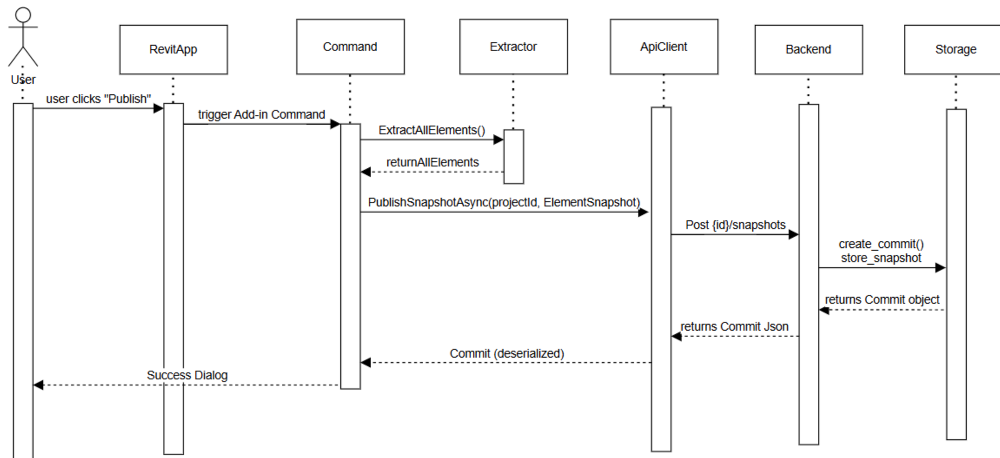
Activity Diagram of Push Changes



Activity Diagram of Pull Changes

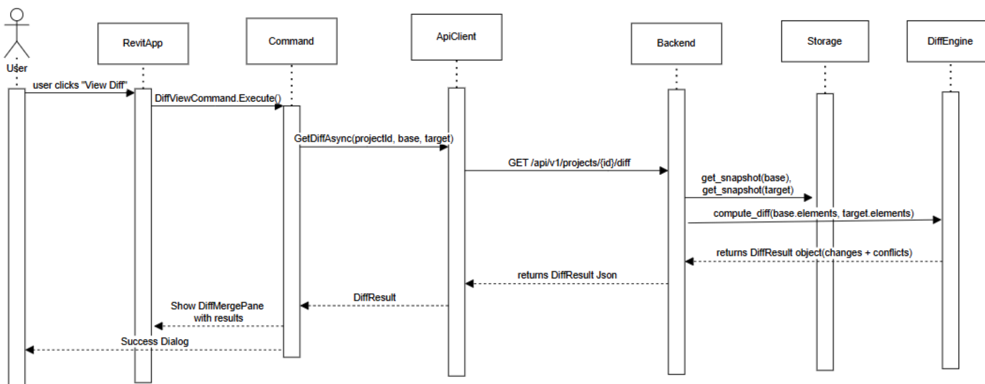
3.6.2.3 Sequence Diagrams

Publish Snapshot

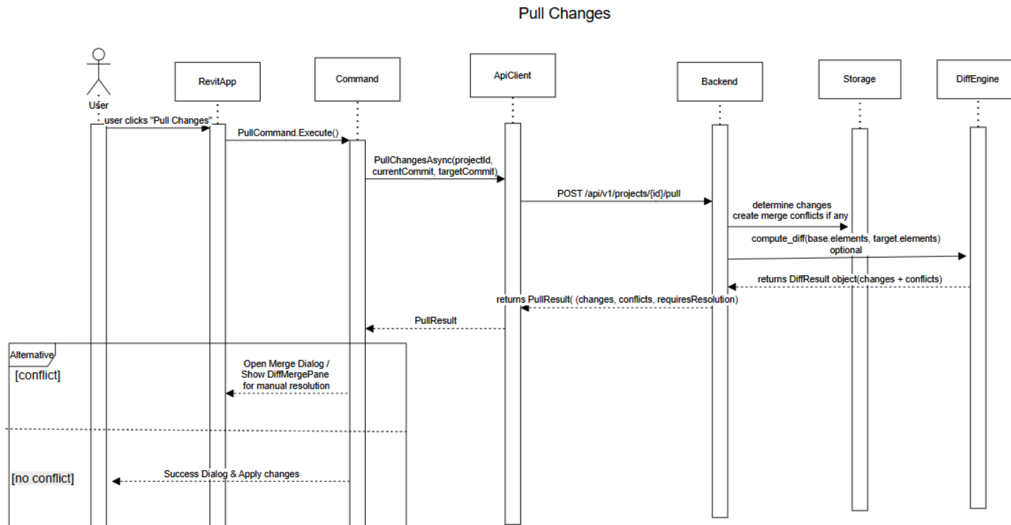


Publish Snapshot (Push) Sequence Diagram

View Diff



View Diff Sequence Diagram



Pull Changes Sequence Diagram

4 Subsystem Services

This section explains the services provided by the subsystems of the proposed system. The system follows a three-layered architecture composed of the GUI Layer, Business Logic Layer, and Data Layer. The GUI Layer contains the user-facing interfaces for authentication, visual difference inspection, version control operations, and commit history viewing. The Business Logic Layer contains the core services that implement routing, authorization, version control, Revit-related model handling, file handling, monitoring, and API communication. The Data Layer provides persistent storage and access to the core entities of the system such as users, projects, memberships, commits, and snapshots.

4.1 GUI Layer

The GUI Layer is composed of four main packages: Account Management Package, Visual Difference Package, Version Control Package, and Commit History Package. These packages contain the interfaces through which the user interacts with the system.

The Account Management Package includes the interfaces related to user authentication and account access. The Login Interface allows existing users to authenticate themselves, while the Registration Interface allows new users to create accounts.

The Visual Difference Package contains the interfaces used for comparing versions and visualizing changes between model states. The Commit Selection Interface allows the user to choose commits for comparison, and the Difference Visualization Interface presents the detected changes and conflicts to the user.

The Version Control Package contains the interfaces for core version control operations. The Push Interface is responsible for uploading a new snapshot and commit to the system. The Pull

Interface is used to fetch changes from another commit or the latest project state. The Merge Interface supports conflict-aware merge operations between revisions.

The Commit History Package contains the Commit History Interface, which enables users to inspect the list of commits and review version history information.

Class / Interface	Description
Login Interface	An interface that allows users to log into the system.
Registration Interface	An interface that allows users to register a new account.
Commit Selection Interface	An interface that allows users to select commits for comparison or version control operations.
Difference Visualization Interface	An interface that displays detected differences, conflicts, and change details between model versions.
Push Interface	An interface that allows users to submit a new commit and snapshot to the system.
Pull Interface	An interface that allows users to retrieve updates from another commit or project state.
Merge Interface	An interface that allows users to merge revisions and resolve conflicts.
Commit History Interface	An interface that displays the version history and commit-related information of a project.

4.2 Business Logic Layer

The Business Logic Layer is responsible for executing the system’s core application logic and coordinating between the GUI Layer and the Data Layer. According to the subsystem decomposition diagram, this layer contains six main packages: Routing & Authorization Package, Revit Core Interaction Package, Version Control Management Package, File Management Package, State Management / Event Monitoring Package, and Communication / API Package.

The Routing & Authorization Package handles authentication and authorization related operations. This functionality is supported by entities such as User, RefreshToken, input/output models like UserRegister, UserLogin, Token, UserRead, and repositories such as UserRepository and TokenRepository.

The Revit Core Interaction Package is responsible for handling model-related domain objects extracted from Revit-based project data. This includes entities such as Element, Geometry, Location, BoundingBox, Point3D, and Parameter. These structures represent model components and their geometric, positional, and parametric properties.

The Version Control Management Package performs the main versioning services of the system. It manages commits, differences, pulls, merges, and conflict resolution. This package is supported by classes such as Commit, CommitBase, CommitCreate, CommitSummary, CommitDetail, DiffEngine, DiffResult, Change, ParameterChange, Conflict, PullRequest, PullResult, MergeRequest, MergeResult, and Resolution.

The File Management Package is responsible for storing and retrieving commit-related files and snapshots. This functionality is represented by the StorageService, which uploads files and resolves stored file paths.

The State Management / Event Monitoring Package coordinates the flow of operations and system state changes during push, pull, diff, and merge processes. It ensures that the correct service is triggered when the user performs an action and that the corresponding result is returned to the interface layer.

The Communication / API Package handles external and internal service communication. It supports requests between the interfaces, business services, repositories, and storage-related components.

Class / Component	Description
Routing & Authorization Package	Handles authentication, authorization, and token-related access control operations.

UserRegister	Stores the input data required for user registration.
UserLogin	Stores the input data required for user login.
Token	Represents the authentication response returned after successful login.
UserRead	Represents user data returned to the client after retrieval.
UserRepository	Provides user creation and retrieval operations.
TokenRepository	Provides refresh token creation, lookup, and deletion operations.
Revit Core Interaction Package	Handles model data structures such as elements, geometry, locations, and parameters.
Element	Represents a model element with category, type, parameters, geometry, and location data.
Geometry	Stores geometric information of an element.
Location	Stores the spatial or transform-based position information of an element.
BoundingBox	Stores minimum and maximum 3D bounds of geometry.
Point3D	Represents a 3D point using x, y, and z coordinates.

Parameter	Represents an element parameter and its metadata.
Version Control Management Package	Handles commit, snapshot, diff, pull, merge, and conflict resolution services.
CommitBase	Stores the common attributes of commit data.
Commit	Represents a stored commit with id, storage location, and optional snapshot.
CommitCreate	Stores the input data needed to create a new commit.
CommitSummary	Represents summarized commit information.
CommitDetail	Represents detailed commit information.
AuthorInfo	Represents simplified author information used in commit outputs.
ElementSnapshot	Represents the full snapshot of model elements at a specific version.
DiffEngine	Computes differences, detects conflicts, and applies selective changes.
DiffResult	Represents the result of a diff operation between two versions.
Change	Represents a detected change on a model element.

ParameterChange	Represents a detected change in a single parameter.
Conflict	Represents an unresolvable or competing change between versions.
PullRequest	Stores the input data required for a pull operation.
PullResult	Represents the output of a pull operation.
Resolution	Stores a chosen conflict resolution for an element.
MergeRequest	Stores the input data required for a merge operation.
MergeResult	Represents the output of a merge operation.
CommitRepository	Provides commit creation, retrieval, listing, and snapshot access operations.
File Management Package	Handles storage and retrieval of files related to commits.
StorageService	Uploads commit files and resolves stored file paths.
State Management / Event Monitoring Package	Tracks operational state changes and outcome statuses during system actions.
Communication / API Package	Transfers data between interface-level requests and backend services.

4.3 Data Layer

The Data Layer is responsible for storing, retrieving, and organizing the persistent data required by the system. According to the subsystem decomposition diagram, this layer consists of Meta Database, Repository, and Entities.

The Entities part contains the core domain objects of the system. Based on the class diagram, these include User, Project, ProjectMember, Commit, RefreshToken, ElementSnapshot, Element, Geometry, Location, Parameter, BoundingBox, Point3D, and the various request/response models used by the application.

The Repository part provides controlled access to persistent data and isolates the business logic from direct database operations. The repositories shown in the class diagram are UserRepository, ProjectRepository, ProjectMemberRepository, TokenRepository, and CommitRepository. These repositories are responsible for creating, querying, and updating the main entities of the system.

The Meta Database stores the persistent records related to users, projects, memberships, refresh tokens, commits, and associated metadata. Commit snapshots and stored files are connected to this layer through repository access and storage management services. This separation allows the rest of the application to evolve without depending directly on the underlying storage details.

Component	Description
Meta Database	Stores the persistent metadata of the system such as users, projects, memberships, tokens, commits, and related records.
Repository	Provides an abstraction layer for reading and writing persistent data.
Entities	Represents the core objects required by the system, including users, projects, commits, snapshots, and model elements.
UserRepository	Handles creation and retrieval of user data.
ProjectRepository	Handles creation and retrieval of project data.

ProjectMemberRepository	Handles project membership, invitation, and membership status operations.
TokenRepository	Handles refresh token creation, lookup, and deletion.
CommitRepository	Handles commit creation, retrieval, listing, counting, and snapshot access.

5 Test Cases

In this section, several test scenarios are introduced to evaluate the behavior of the system after implementation. These tests aim to confirm that the main features of the application function correctly and that the system performs reliably. The results of these tests will be examined and included in the final report.

5.1 Functional Test Cases

Test ID: F-TC-01

Test Type/Category: Functional, Integration, Security

Summary / Title / Objective:
Verify that a user can successfully authenticate through the CollabHub plugin before accessing backend services.

Priority / Severity: Critical

Procedure of Testing Steps:

Action	Expected Result
User opens the CollabHub login interface	Login screen appears
User enters valid credentials	System authenticates the user
Plugin sends authentication request to backend	Backend verifies credentials
Backend returns authentication response	Access token/session is returned

User accesses plugin features	Protected features become available
User enters incorrect credentials	System denies access and displays an error message

Test ID: F-TC-02

Test Type/Category: Functional, Authentication

Summary / Title / Objective:
Verify that a new user can successfully register in the system and obtain authentication tokens.

Priority / Severity: Critical

Procedure of Testing Steps:

Action	Expected Result
User opens the registration interface	Registration form appears
User enters name, email, and password	Input fields accept data
User submits the registration request	Backend creates the user account
Backend processes registration	Authentication tokens are returned
Plugin receives response	User becomes logged in automatically

Test ID: F-TC-03

Test Type/Category: Functional, Integration

Summary / Title / Objective:
Verify that a new project can be initialized from an existing Revit model.

Priority / Severity: Critical

Procedure of Testing Steps:

Action	Expected Result
User logs into the CollabHub plugin	Authentication succeeds

User opens a Revit model file	Model loads successfully
User clicks the Create Project button	Project creation dialog appears
User enters project information	Form accepts project name
User confirms project creation	Backend creates a new project
Plugin sends the base model file	Base file is uploaded successfully
Initialization finishes	Project becomes tracked by CollabHub

Test ID: F-TC-04

Test Type/Category: Functional, Component

Summary / Title / Objective:
Verify that the plugin can extract element data from a Revit model for snapshot creation.

Priority / Severity: Critical

Procedure of Testing Steps:

Action	**Expected Result**
User opens a tracked Revit model	Model loads successfully
User modifies architectural elements	Local changes are made
User saves the document	Changes are stored locally
User triggers the publish operation	Plugin starts extraction process
Plugin reads elements from Revit API	Element data is collected
Snapshot data is prepared	Extracted data is ready to send to backend

Test ID: F-TC-05

Test Type/Category: Functional, Integration

Summary / Title / Objective:
Verify that model changes can be pushed to the backend as a new version.

Priority / Severity: Critical

Procedure of Testing Steps:

Action	Expected Result
User modifies elements in a tracked project	Model contains unsynchronized changes
User clicks the Publish/Push button	Snapshot creation begins
Plugin compares current state with previous commit	Differences are identified
Plugin sends snapshot to backend	Backend processes the request
Backend creates a new commit entry	Snapshot and metadata are stored
Operation completes	User receives success message

Test ID: F-TC-06

Test Type/Category: Functional, Integration

Summary / Title / Objective:
Verify that users can retrieve and view commit history of a project.

Priority / Severity: Major

Procedure of Testing Steps:

Action	Expected Result
User opens a tracked project	Project loads normally
User opens the History panel	Commit list is displayed
Plugin requests commit history from backend	Backend returns commit data
History panel loads the results	Commits appear with timestamps and authors
User selects a commit entry	Detailed information is displayed

Test ID: F-TC-07

Test Type/Category: Functional, Integration

Summary / Title / Objective:
Verify that the system can compute the differences between two commits.

Priority / Severity: Major

Procedure of Testing Steps:

Action	Expected Result
User selects two commits from history	Commit IDs are selected
Plugin sends diff request to backend	Backend receives comparison request
Backend loads snapshot data	Snapshots are retrieved successfully
Backend computes differences	Diff summary is generated
Results are returned to plugin	Change information is displayed

Test ID: F-TC-08

Test Type/Category: Functional, Integration

Summary / Title / Objective:
Verify that a user can pull updates from the backend to synchronize their local model.

Priority / Severity: Critical

Procedure of Testing Steps:

Action	Expected Result
Another collaborator publishes a new version	New commit appears on server
User opens local project	Local version is outdated
User clicks Pull Updates	Plugin sends pull request
Backend computes changes	Diff between commits is generated
Backend returns update data	Plugin receives update information

Plugin applies changes to model	Local model becomes synchronized

Test ID: F-TC-09

Test Type/Category: Functional, Collaboration

Summary / Title / Objective:
Verify that a project owner can invite another user to collaborate on a project.

Priority / Severity: Major

Procedure of Testing Steps:

Action	Expected Result
Project owner opens collaborator settings	Invitation interface appears
Owner enters collaborator email	Email is validated
Owner sends invitation	Backend creates invitation entry
Invited user logs in	Pending invitations appear
Invited user accepts invitation	Backend activates collaborator access
User downloads base project file	Project becomes accessible to collaborator

5.2 Non-functional Test Cases

Test ID: NF-TC-01

Test Type/Category: Performance

Summary / Title / Objective:
Evaluate the performance of snapshot publishing during normal usage.

Priority / Severity: Major

Procedure of Testing Steps:

Action	Expected Result
User modifies a small number of elements	Local changes are present

User starts publish operation	Snapshot extraction begins
Plugin processes data	Operation completes without freezing
Backend stores commit	Publish completes in under 15 seconds

Test ID: NF-TC-02

Test Type/Category: Reliability

Summary / Title / Objective:
Verify that corrupted or incomplete snapshot data is rejected by the backend.

Priority / Severity: Critical

Procedure of Testing Steps:

Action	**Expected Result**
Send malformed snapshot data to backend	Backend receives invalid data
Backend performs validation	Request is rejected
System logs the error	No invalid commit is stored
Retry with valid data	Request succeeds normally

Test ID: NF-TC-03

Test Type/Category: Security

Summary / Title / Objective:
Ensure that protected API endpoints require authentication.

Priority / Severity: Critical

Procedure of Testing Steps:

Action	**Expected Result**
Send request without authentication token	Backend rejects request
Send request with valid token	Request is accepted

Send request with invalid token	Access is denied

Test ID: NF-TC-04

Test Type/Category: Reliability

Summary / Title / Objective:
Check system behavior when network connection fails during synchronization.

Priority / Severity: Critical

Procedure of Testing Steps:

Action	**Expected Result**
Start a push operation	Upload begins
Disable network connection	Communication is interrupted
Plugin detects failure	Error message is displayed
Restore connection	Operation can be retried successfully

Test ID: NF-TC-05

Test Type/Category: Compatibility

Summary / Title / Objective:
Verify that the CollabHub plugin loads correctly inside Autodesk Revit.

Priority / Severity: Major

Procedure of Testing Steps:

Action	**Expected Result**
Install the plugin files	Plugin files are available
Launch Autodesk Revit	Revit starts normally
Check plugin panel	CollabHub panel appears
Perform simple action	Plugin works correctly inside Revit

Test ID: NF-TC-06

Test Type/Category: Installation

Summary / Title / Objective:
Verify correct plugin installation and startup.

Priority / Severity: Major

Procedure of Testing Steps:

Action	Expected Result
Install plugin files in add-in directory	Installation completes
Start Revit	Plugin loads automatically
Open plugin panel	Interface becomes visible

Test ID: NF-TC-07

Test Type/Category: Maintainability

Summary / Title / Objective:
Verify that system logs and documentation allow debugging.

Priority / Severity: Minor

Procedure of Testing Steps:

Action	Expected Result
Perform push operation	Event recorded in logs
Perform pull operation	Operation logged
Trigger system error	Error message appears in logs

Test ID: NF-TC-08

Test Type/Category: Data Integrity

Summary / Title / Objective:
Verify that commit history remains consistent after multiple publishes.

Priority / Severity: Major

Procedure of Testing Steps:

Action	Expected Result
Publish first snapshot	Commit created
Publish additional snapshots	New commits created
Request commit history	All commits appear in order
Inspect commit metadata	Parent relationships are correct

6 Consideration of Various Factors in Engineering Design

6.1 Constraints

6.1.1 Implementation Constraints

- CollabHub will be developed primarily as a Autodesk Revit Plugin using C# and the .NET framework to ensure integration with the Revit environment.
- The backend will be built using Python (specifically with FastAPI framework) as it is a framework that most of the team are experienced with.
- PostgreSQL will be used as the relational database for storing metadata (commit logs, user info, branch pointers).
- The system must comply with the Autodesk Revit API limitations; specifically, it must handle main-thread restrictions when interacting with the active model to prevent freezing the user interface.[4]

6.1.2 Economic Constraints

- The project must utilize open-source libraries and free-tier cloud services where possible to fit within a project budget of zero.
- Since .rvt files are significantly larger (Gigabytes) than text files of code, Cloud storage costs are a major constraint. The system must implement an efficient way of storing the .rvt files in its database.
- The development requires access to Autodesk Revit licenses. The team relies on educational licenses that can be accessed through university..

6.1.3 Ethical Constraints

- **Intellectual Property (IP) Protection:** Architects' designs are their trade secrets. The system must ensure that proprietary design files are not accessible to unauthorized users.
- **Data Integrity:** In the construction industry, a corrupted file can lead to structural errors in the real world. The system is ethically bound to verify the integrity of every uploaded and downloaded file to prevent losses and potential safety hazards.(eg. The system must clearly inform the user if a "merge" operation resulted in any data loss or if a conflict requires manual intervention.)

- **Privacy:** User activity logs must be stored securely and only visible to authorized team members, not exposed publicly.

6.1.4 Social Constraints

- **Usability for Non-Programmers:** Unlike software engineers, architects may not be familiar with usage of command line tools. The application must provide a Graphical User Interface (GUI) that abstracts complex version control concepts into understandable terms.
- **Collaboration:** The system should foster a collaborative environment by allowing multiple architects to work on different parts of a building simultaneously without "locking" the entire project. This constraint also reflects the core functionality this project aims to achieve.

6.2 Standards

IEEE 830 The IEEE 830 standard provides the guidelines for our Software Requirements Specifications (SRS). By following this standard, we ensure that our requirements for "RVT Versioning" and "Visual Diffing" are clearly defined, unambiguous, and verifiable, serving as a solid contract between the developers and the stakeholders.[3]
ISO 19650 (BIM Information Management) While primarily a construction standard, ISO 19650 defines the concepts of a Common Data Environment (CDE). CollabHub aligns with these standards by managing the "Work in Progress" and "Shared" states of information containers (RVT files), ensuring the project adheres to industry-standard information lifecycles.[5]
UML 2.5.1 - Unified Modeling Language We utilize UML 2.5.1 to visualize the complex interactions of our system. Specifically, Sequence Diagrams are used to map the synchronization process between the Client Plugin and the API Gateway, and Class Diagrams are used to structure the internal logic of the proprietary file parser.[6]
AES-256 (Advanced Encryption Standard): All data storage will be encrypted using AES-256 to prevent unauthorized access [2].

7 Teamwork Details

7.1 Contributing and Functioning Effectively on the Team

We developed CollabHub as a five-member team by dividing responsibilities based on the needs of the project. Moin Khan implemented the core functionality, including push, pull, and Revit API related features. Yiğit Özhan worked on the UI and backend connections, helping ensure correct communication between system components. İbrahim Çaycı handled project initialization, login, registration, authentication, and repository access together with Tuna Göksal. Ömer Edip Aras worked on database creation, entities, and backend connections. Tuna Göksal also contributed to repository access and improved the push and pull functionalities.

To function effectively, we coordinated tasks regularly and stayed in communication throughout development. Since the Revit plugin, backend, and database layers were closely connected,

we made sure our work remained compatible and integrated properly. This helped us reduce overlap, solve issues earlier, and maintain steady progress.

7.2 Helping create a collaborative and inclusive environment

We maintained a collaborative working environment by discussing important decisions together and keeping communication active throughout the project. Although each member had primary responsibilities, we shared progress, discussed problems, and supported each other when tasks overlapped.

We also made sure that every team member stayed informed about the general state of the system, not only their own part. This created a more inclusive process and helped us make decisions with a better understanding of the whole project. Open discussion and mutual feedback improved both teamwork and system consistency.

.

7.3 Taking lead role and sharing leadership on the team

Leadership in our team changed depending on the part of the project we were working on. For example, Moin Khan took the lead in the core functionality of CollabHub, especially in push, pull, and Revit API related features, since these were central to the main purpose of the system. Yiğit Özhan took responsibility in UI and backend communication tasks, helping make sure that the interface worked properly with the backend. İbrahim Çaycı led the project setup and authentication side by handling project initialization, login, registration, and authorization. In repository access features, İbrahim Çaycı and Tuna Göksal shared responsibility, while Tuna Göksal also took initiative in improving push and pull functionalities. Ömer Edip Aras led the data layer work through database creation, entity design, and backend connections.

Even though each member had areas where they took more responsibility, leadership was not limited to individual tasks. When integration problems or system-wide decisions came up, we discussed them together and decided as a team. This made leadership more balanced and helped us keep the project coordinated.

8 Glossary

AES-256 (Advanced Encryption Standard): A symmetric encryption algorithm that uses a 256-bit key to secure data. In this project, it is used to encrypt RVT files stored on the server to protect intellectual property.

BIM (Building Information Modeling): A process involving the generation and management of digital representations of physical and functional characteristics of places. CollabHub is a tool designed specifically to manage BIM data.

Clash Detection: An automated process in architectural software that identifies where two building elements (e.g., a pipe and a wall) physically overlap or interfere with one another.

CORS (Cross-Origin Resource Sharing): A security feature that allows the Revit Plugin (client) to make requests to the FastAPI backend domain.

Diffing (Differential): The process of comparing two versions of a file or object to identify changes. In CollabHub, this refers specifically to identifying geometric or parameter changes in architectural elements.

ElementSnapshot: A structured JSON object defined in this project that represents the state of a single Revit element (geometry and parameters) at a specific point in time, used for comparison by the Diff Engine.

Geometry Hash: A computed alphanumeric string generated from the vertex data and dimensions of a 3D object. If the hash changes, the system knows the geometry has been modified.

Revit API: The Application Programming Interface provided by Autodesk. It allows the CollabHub plugin to programmatically read, extract, and modify 3D elements within an active Revit session.

Ribbon: The main toolbar interface in Autodesk Revit where the CollabHub plugin buttons (Push, Pull, Login) will be located.

RVT (.rvt): The proprietary file extension for Autodesk Revit project files, which contain the full architectural model and metadata.

9 References

[1] N. Leavitt, “Software Version Control Systems,” Computer, vol. 38, no. 6, 2005.

[2] National Institute of Standards and Technology, “Advanced Encryption Standard (AES),” FIPS PUB 197, 2001.

[3] IEEE, “IEEE Std 830-1998: Recommended Practice for Software Requirements Specifica tions,” 1998.

[4] Autodesk, "Revit API Developers Guide," Autodesk Knowledge Network, 2024. [Online]. Available: https://help.autodesk.com/view/RVT/2024/ENU/

[5] Organization and digitization of information about buildings and civil engineering works, including building information modelling (BIM) — Information management using building information modelling — Part 1: Concepts and principles, ISO 19650-1:2018, Dec. 2018.

[6] Object Management Group, "Unified Modeling Language (UML) Specification Version 2.5.1," Dec. 2017. [Online]. Available: https://www.omg.org/spec/UML/2.5.1/PDF