Bilkent University
Department of Computer Engineering

Senior Design Project
T2527
CollabHub

Final 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

Contents

1 Introduction **3**
2 Requirements Details **3**
2.1 Functional Requirements 3
2.2 Non-Functional Requirements 4
3 Final Architecture and Design Details **5**
3.1 Subsystem Decomposition 5
3.2 Hardware/Software Mapping 7
3.3 Persistent Data Management **8**
3.4 Access Control and Security 8
3.5 System Models **10**
3.5.1 Use-Case Model **10**
3.5.2 Object and Class Model **11**
3.5.2.1 Class Diagram 11
3.5.2.2 Activity Diagrams 11
3.5.2.3 Sequence Diagrams 14
4 Development / Implementation Details **15**
4.1 Frontend (Revit Add-in) 15
4.2 Backend 16
4.3 Database 16
4.4 Summary of Workflow 17
5 Test Cases and Results **18**
5.1 Functional Test Cases 18
5.2 Non-functional Test Cases 30
6 Maintenance Plan and Details **36**
7 Other Project Elements **36**
7.1 Consideration of Various Factors in Engineering Design 36
7.1.1 Constraints 36
7.1.1.1 Implementation Constraints 36
7.1.1.2 Economic Constraints 37
7.1.1.3 Ethical Constraints 37
7.1.1.4 Social Constraints 37
7.1.2 Standards 37
7.2 Ethics and Professional Responsibilities 38
7.3 Teamwork Details **38**
7.3.1 Contributing and Functioning Effectively on the Team 38
7.3.2 Helping create a collaborative and inclusive environment 39
7.3.3 Taking lead role and sharing leadership on the team 39
7.3.4 Meeting Objectives 39
7.3.5 Problems Encountered and Our Solutions 40
7.4 New Knowledge Acquired and Applied 40

8 Conclusion and Future Work **40**
9 Glossary **41**
10 References **42**

1 Introduction

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, compare different versions of a model, use branching and merging for a better flow of collaboration. By focusing on the structure of RVT files and transmitting only the differences between versions, CollabHub provides a more efficient and accessible solution for architectural collaboration.

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.

2 Requirements Details

2.1 Functional Requirements

The following requirements define the specific behaviors and functions the system must support:

1. The system extracts all relevant architectural data from the project file and converts it into a structured format suitable for backend processing.
2. The system supports uploading the extracted base version of the file to the backend, establishing the initial reference state for future comparisons.
3. The client compares the current file state with the previously stored version to identify additions, modifications, and deletions.
4. Only the identified differences are transmitted to the backend instead of resending the entire project file.
5. Each new version will be stored on the backend together with basic metadata such as timestamp and user information.
6. The client will allow users to pull the latest available version from the backend and apply the corresponding changes to their local file.
7. When a change cannot be applied (e.g., due to missing or altered referenced components), the system will notify the user of the conflict.
8. Where supported, the client will utilize built-in visual comparison tools to highlight differences between versions.

9. A custom visual comparison tool may be developed in later stages, but it is not a requirement for the current phase of the project.
10. The client interface will provide simple controls that allow the user to push changes, pull updates, and view available versions within the application.

2.2 Non-Functional Requirements

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.

3 Final Architecture and Design Details

3.1 Subsystem Decomposition

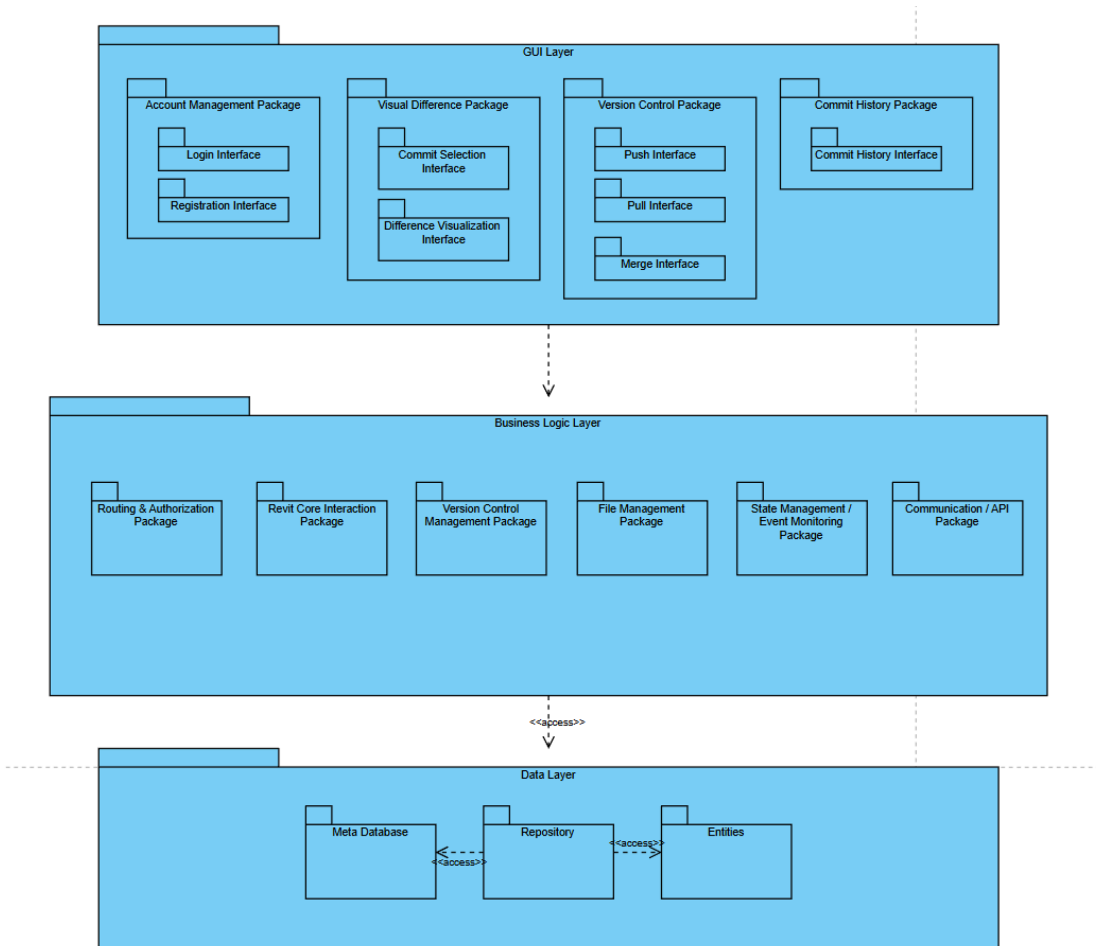
The CollabHub system architecture is decomposed into three distinct layers to manage the processing and bandwidth requirements of 3D architectural files. This structure separates the user interface, core processing logic, and persistent storage mechanisms.

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.

.



3.2 Hardware/Software Mapping

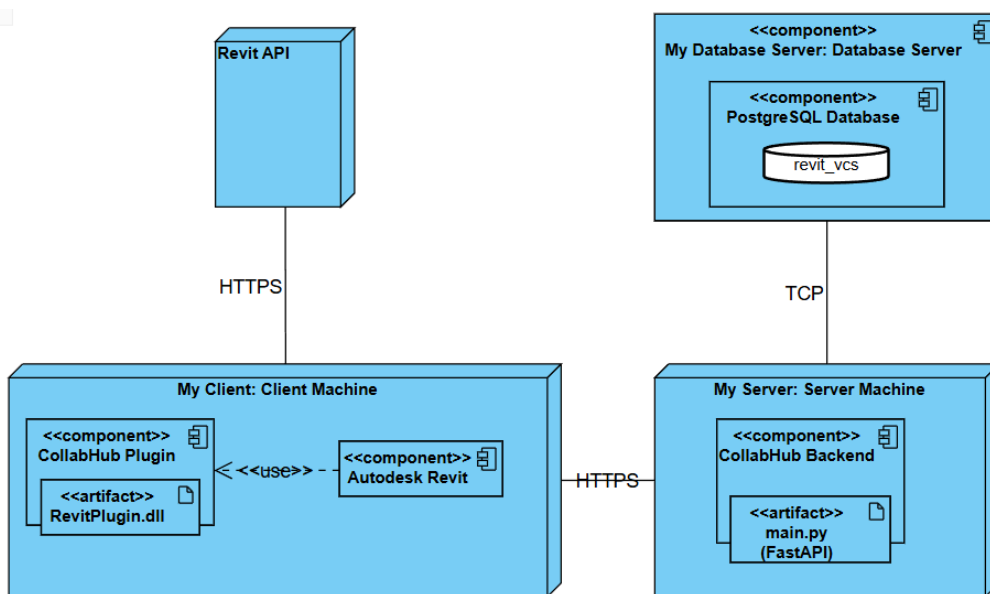


Figure: Deployment Diagram

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.

3.3 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.

3.4 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.

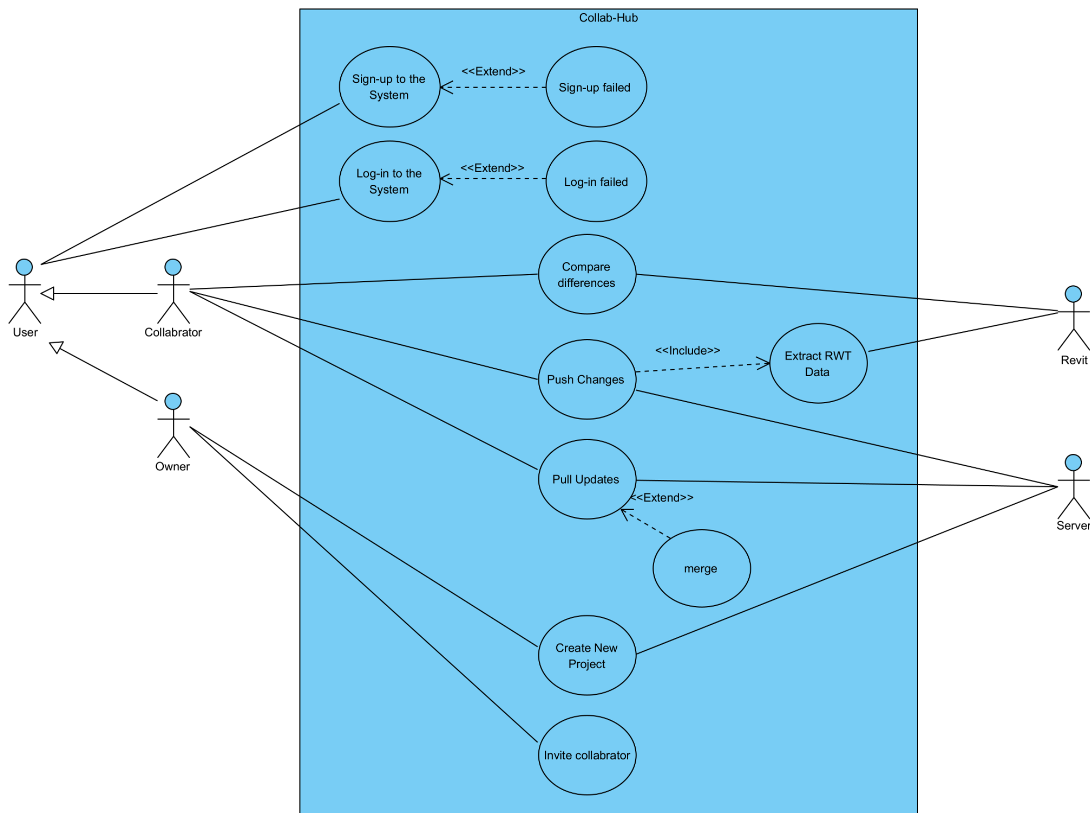
Data stored on the server is also protected through secure database access. The backend communicates with the PostgreSQL database using controlled queries, ensuring that only the backend

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.

3.5 System Models

3.5.1 Use-Case Model



3.5.2 Object and Class Model

3.5.2.1 Class Diagram

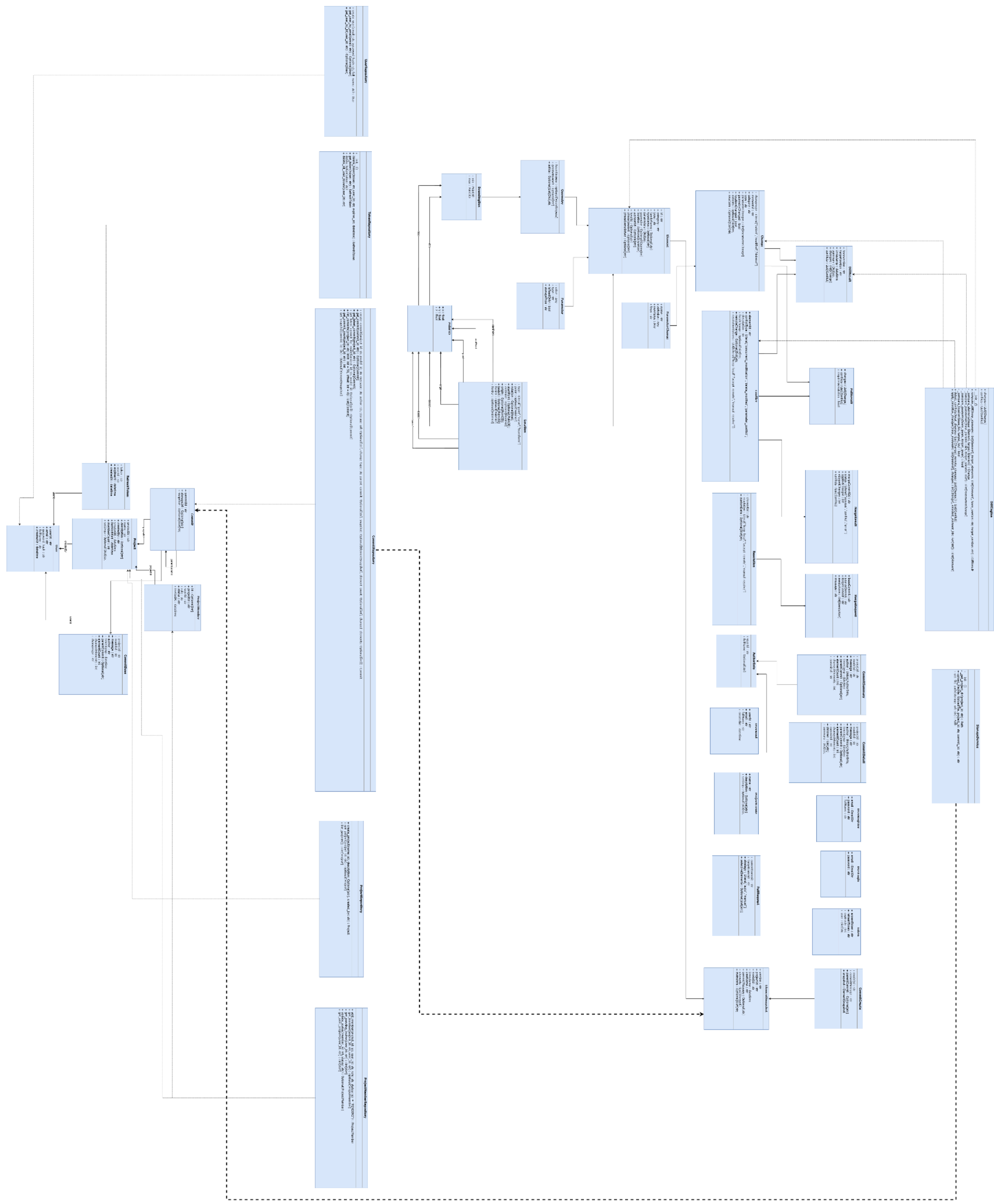
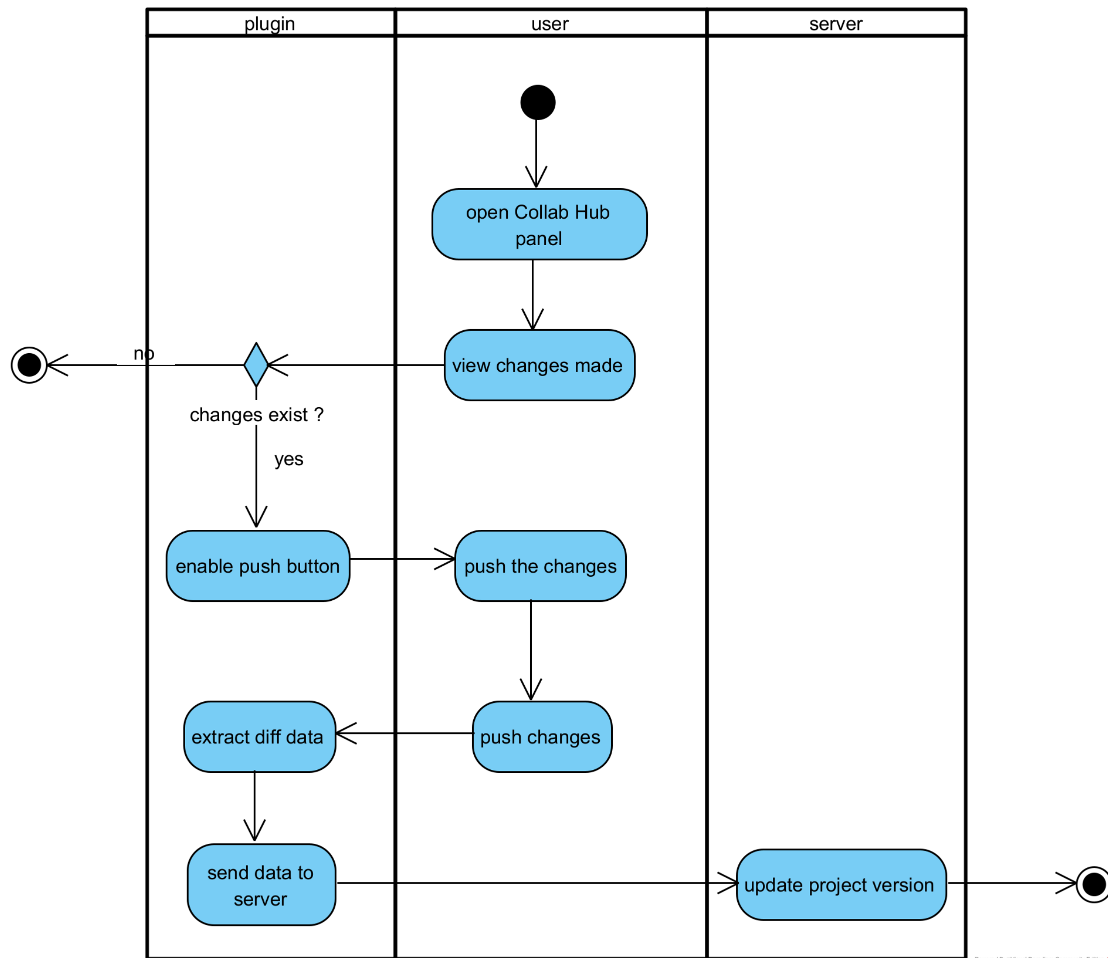
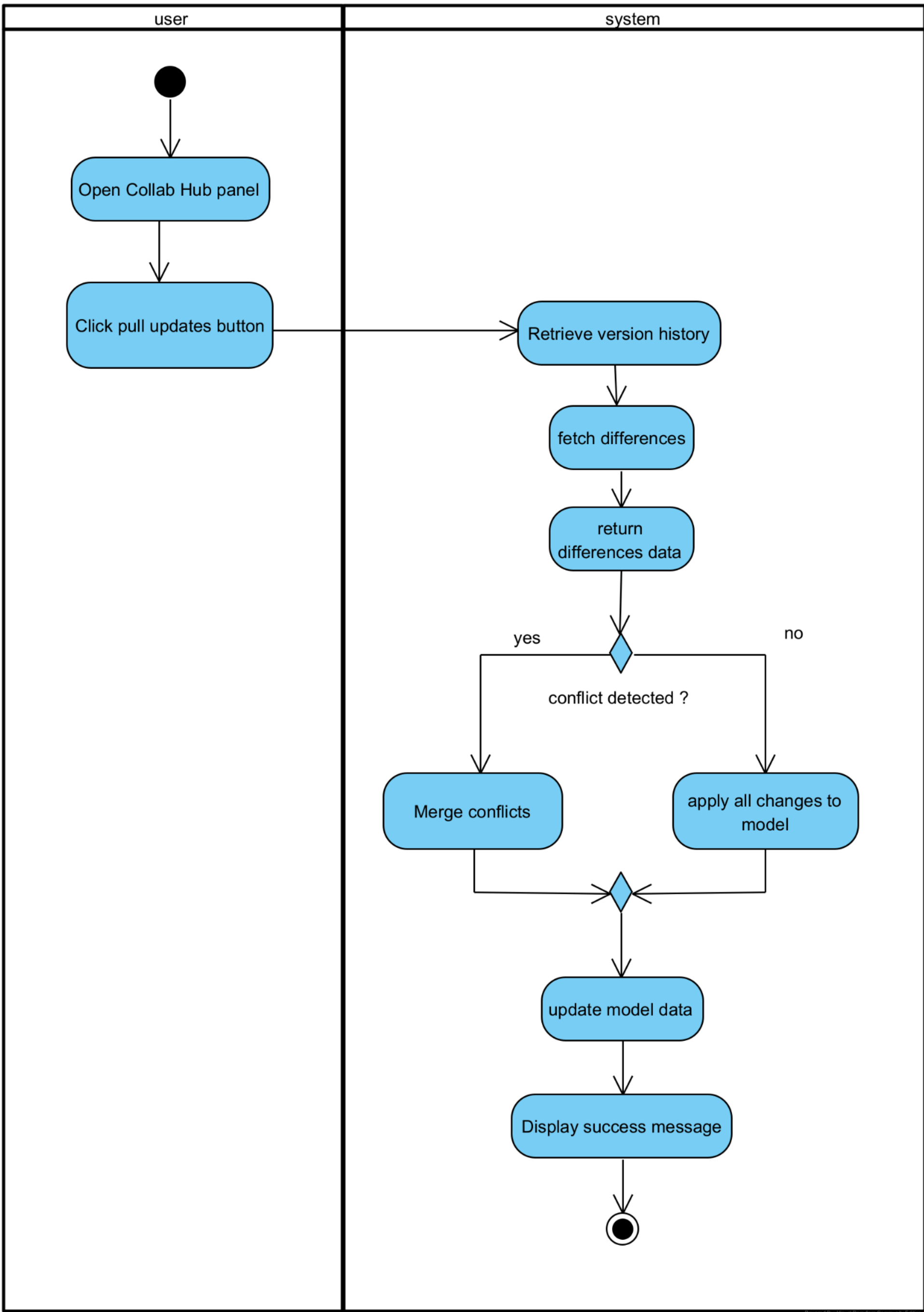


Figure : Class & Object Diagram

3.5.2.2 Activity Diagrams

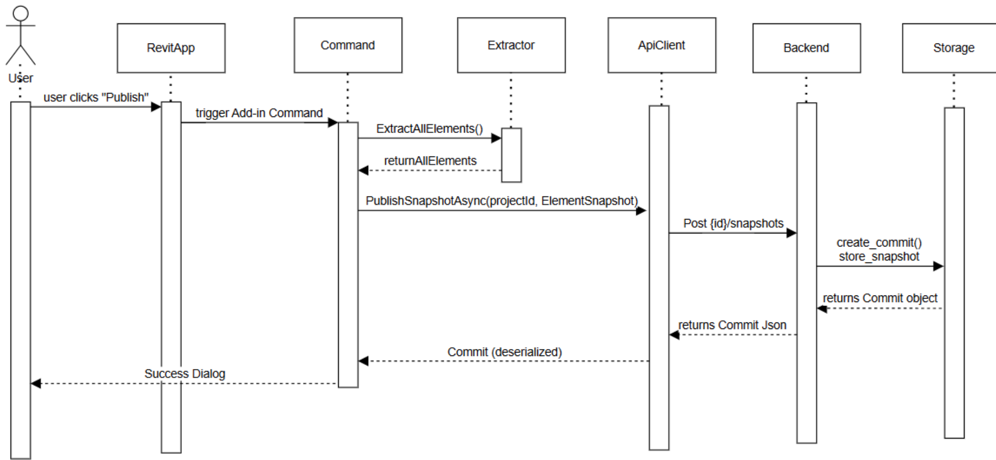


Activity Diagram of Push Changes

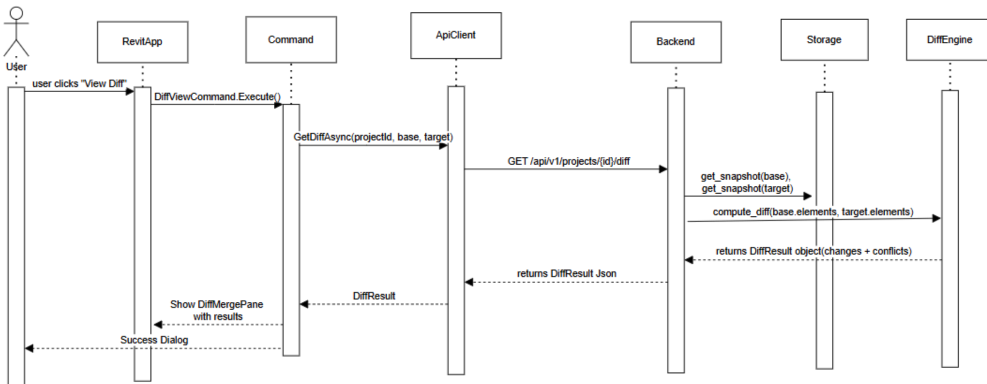


Activity Diagram of Pull Changes

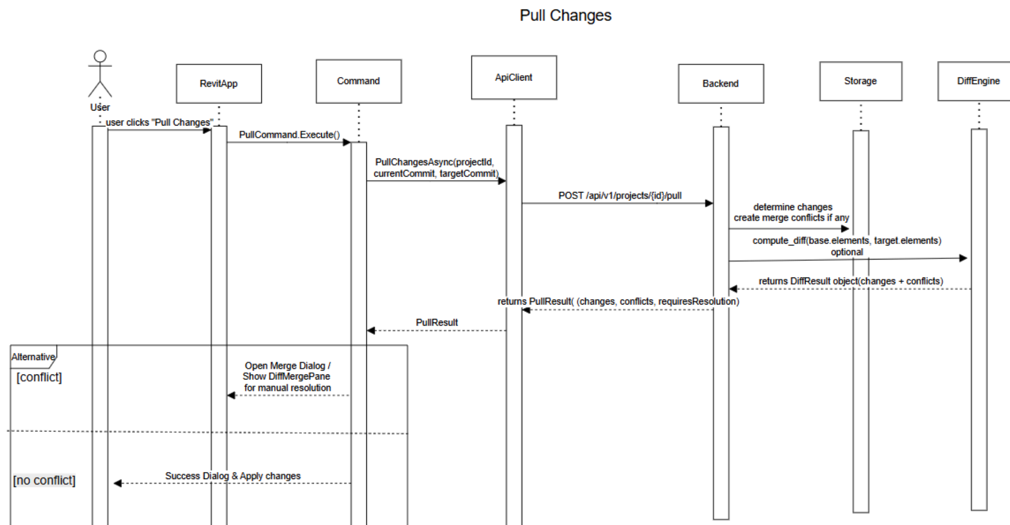
3.5.2.3 Sequence Diagrams



Publish Snapshot (Push) Sequence Diagram



View Diff Sequence Diagram



Pull Changes Sequence Diagram

4 Development / Implementation Details

4.1 Frontend (Revit Add-in)

The frontend of CollabHub is implemented as a Revit add-in using C# with .NET Framework 4.8 and WPF (Windows Presentation Foundation) for the user interface. It interfaces directly with the Revit API 2026 to access and manipulate building elements within Revit projects. The add-in architecture is built around external commands that implement the IExternalCommand interface, which Revit uses to invoke plugin functionality. Key commands include the LoginCommand for handling user authentication and logout operations, the RegisterCommand for new user registration, and various version control operations such as publish, commit, merge, and diff functions.

The user interface is constructed using WPF dialogs and sidebar panels. The LoginDialog provides the authentication interface where users enter credentials, while the InvitationsDialog manages project invitation workflows. Two main provider classes manage persistent UI state: HistoryPaneProvider displays the version history of projects and elements in a sidebar panel, and DiffMergePaneProvider provides visualization and interaction for comparing versions and performing merge operations. These panels persist across Revit sessions, giving users continuous access to version control features.

Behind the scenes, several service classes handle the technical operations. The ApiClient service manages all HTTP communication with the FastAPI backend, handling authentication tokens and request serialization. The ElementApplier service contains the logic for applying merge changes back into the Revit document, translating backend merge instructions into actual element modifications. Additional services handle the extraction of Revit element properties and their serialization into JSON format suitable for transmission to the backend.

The build process is configured to automatically deploy the compiled add-in. A post-build event copies the generated DLL and the .addin manifest file to %AppData%\Autodesk\Revit\Addins\2026\, which is the standard

location where Revit searches for plugins. This automation streamlines development and ensures the add-in is available for testing immediately after building.

4.2 Backend

The backend is built using FastAPI, a modern Python web framework, and runs on an Uvicorn ASGI server. The API is accessible at http://localhost:8000 and is organized around RESTful endpoints documented automatically through interactive Swagger UI at /api/docs and ReDoc at /api/redoc. The architecture follows clean layering principles with routers handling HTTP endpoints, services implementing core business logic, entities defining database models, repositories managing data access, and schemas validating request and response payloads using Pydantic.

The authentication system is implemented through JWT tokens generated using the python-jose library. The auth router handles user login, registration, and token validation, with passwords hashed using bcrypt through the passlib library. This allows the add-in to authenticate once and then use JWT tokens for subsequent API calls without resending credentials.

The core functionality is distributed across several routers. The projects router manages the creation and retrieval of projects, which serve as containers for version control work. The branches router handles named branches within projects, allowing parallel work streams. The snapshots router stores element state at specific points in time, capturing the complete geometry and parameter data for elements. The diff router compares snapshots to detect what has changed between versions, using the powerful DiffEngine service to perform this analysis. The merge router handles the logic for combining changes from different branches and resolving conflicts. Supporting routers manage base RVT file metadata through the base_files router and handle data uploads from the add-in through the payloads router.
The DiffEngine is the core analytical component of the backend. It accepts two snapshots of elements from different versions and computes detailed differences, including which elements were added, deleted, or modified. For modified elements, it identifies which specific parameters changed and their old and new values. The engine also detects conflicts that occur when the same parameter is modified differently in two branches being merged. The OperationEngine complements this by applying merge operations, determining which changes should be applied when merging branches and handling the transformation logic.
The project uses SQLModel as its Object-Relational Mapping (ORM) layer, which integrates SQLAlchemy with Pydantic validation. This provides a modern approach to database interaction that combines robust database access with strong type checking and validation. While the architecture supports PostgreSQL as the target database, the current implementation stores data in memory during a session, making it suitable for development and testing but requiring the persistent database layer to be implemented for production use.

4.3 Database

The database layer of CollabHub currently operates with in-memory storage during development, though the architecture is designed to support PostgreSQL as the production database. The data modeling is handled through SQLModel entities, which provide a clean way to define both database schema and API validation rules simultaneously. This approach reduces duplication and ensures consistency between the data model and the API contracts.

The system defines several key entities that represent the core concepts of version control. The User entity stores authentication credentials and user profile information. The Project entity represents a version control project that acts as a container for collaborative work. The ProjectMember entity links users to projects and stores role information, enabling role-based access control. The Branch entity represents named development lines within a project, allowing multiple parallel versions to exist simultaneously. The Commit entity captures snapshots of the project state at specific moments, with each commit linked to a branch and containing metadata about when it was created and by whom. The Token entity stores JWT authentication tokens with expiration times. Finally, the Element entity represents individual Revit building elements with all their parameters and geometric properties in a serialized format.

The data flow through the system begins when a user working in Revit publishes changes. The add-in extracts element properties from the Revit document and serializes them to JSON format. This serialized snapshot is sent to the backend's snapshots endpoint and stored either in memory or in the database as a new commit on the current branch. When a user wants to understand what has changed between versions, they initiate a diff operation. The backend retrieves the snapshots for both versions, passes them to the DiffEngine, and returns a detailed report of all changes. When the user is ready to merge changes from another branch, the backend applies selective operations through the OperationEngine, determining which changes to accept or reject based on user selections. The resulting merge commit is stored, and the add-in is notified so it can update the RVT document with the approved changes.

For development and testing purposes, the system pre-seeds the database with test users having the email addresses user1@gmail.com and user2@gmail.com. This allows developers to test authentication workflows and multi-user scenarios without manual setup. However, this seeding mechanism is explicitly intended to be removed before potential production deployment in future to avoid security issues and unnecessary test data in live systems.

4.4 Summary of Workflow

The complete workflow of CollabHub demonstrates how the frontend, backend, and data layer work together. Initially, a user launches the Revit add-in and authenticates through the LoginDialog by entering their credentials. The add-in sends these credentials to the backend's authentication endpoint, receives a JWT token in response, and stores it for future API calls. The add-in updates its UI to reflect the logged-in state and populates the HistoryPaneProvider with the user's projects and branches.

As the user makes changes to their Revit model, they can publish these changes through an add-in command. The add-in extracts the properties of all relevant elements, serializes them, and sends them to the backend as a new snapshot on the current branch. The backend stores this snapshot as a commit, creating a permanent record of the model state at that moment. When the user wants to understand what changed between their current work and another branch, they request a diff. The backend retrieves both snapshots, processes them through the DiffEngine, and returns a detailed comparison showing added elements, deleted elements, modified elements, and specific parameter changes. This information is displayed in the DiffMergePaneProvider, allowing the user to review exactly what differs.
When the user decides to merge changes from another branch into their current work, they initiate a merge operation through the add-in. The backend's merge logic combines the snapshots and uses the DiffEngine to identify potential conflicts. The add-in presents these changes and conflicts to the user for selective approval. Once the user selects which changes to accept, the OperationEngine generates a set of operations that transform the current model state into the merged state. The add-in receives these operations and uses the

ElementApplier service to apply them to the active Revit document, updating element properties and creating or deleting elements as specified. A final commit is created to record the merged state, completing the cycle.

5 Test Cases and Results

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.

All 27 test cases, 18 functional and 9 non-functional, were executed on the final version of CollabHub and all of the implemented parts were passed successfully. However merging feature is not yet completed, so the tests related to merging (4 tests) are not successful. This feature will be completed according to the maintenance plan in the future..The results confirm that the system correctly handles authentication, project initialization, push and pull operations, diff computation, collaboration workflows, and non-functional requirements including performance, security, reliability, and compatibility.

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

Result : Final version of the program successfully passes this test

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

Result : Final version of the program successfully passes this test

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 Initialize Project button	Project initialization 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

Result : Final version of the program successfully passes this test

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

Result : Final version of the program successfully passes this test

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

Result : Final version of the program successfully passes this test

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

Result : Final version of the program successfully passes this test

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

Result : Final version of the program successfully passes this test

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

Action	Expected Result
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

Result : Final version of the program successfully passes this test

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

Result : Final version of the program successfully passes this test

Test ID: F-TC-10

Test Type/Category: Functional, Integration

Summary / Title / Objective:
Verify that a user can create a new branch from an existing project commit .

Priority / Severity: Critical

Procedure of Testing Steps:

Action	Expected Result
User opens a tracked project	Project loads successfully
User opens the Branch management panel	Branch list is displayed
User selects a base commit	Commit is highlighted as branch point
User clicks Create Branch	Branch creation dialog appears
User enters a branch name	Input field accepts the name
User confirms creation	Backend creates the new branch
Plugin receives confirmation	New branch appears in branch list

Result : Final version of the program successfully passes this test

Test ID: F-TC-11

Test Type/Category: Functional, Integration

Summary / Title / Objective:
Verify that a user can switch between branches and that the local model updates accordingly .

Priority / Severity: Critical

Procedure of Testing Steps:

Action	Expected Result
User opens the Branch panel	Available branches are listed
User selects a different branch	Switch confirmation prompt appears
User confirms the switch	Plugin sends branch switch request to backend

Backend returns the latest snapshot for that branch	Plugin receives branch data
Plugin applies the branch state to the local model	Local model reflects the selected branch
User verifies model content	Content matches the selected branch

Result Final version of the program successfully passes this test

Test ID: F-TC-12

Test Type/Category: Functional, Integration

Summary / Title / Objective:
Verify that a user can publish changes on a branch without affecting the main branch.   .

Priority / Severity: Critical

Procedure of Testing Steps:

Action	Expected Result
User switches to a feature branch	Branch switch succeeds
User modifies elements in the Revit model	Local changes are present
User clicks Publish/Push	Snapshot is created for the active branch
Backend creates a new commit on that branch	Commit is stored under the correct branch
User switches back to main branch	Main branch state is unchanged
User inspects main branch history	Branch commit does not appear in main

Result : Final version of the program successfully passes this test

Test ID: F-TC-13

Test Type/Category: Functional, Integration

Summary / Title / Objective:
Verify that a user can initiate a merge from a feature branch into the main branch . .

Priority / Severity: Critical

Procedure of Testing Steps:

Action	Expected Result
User opens the Branch panel	Branch list is displayed
User selects the source branch to merge from	Source branch is highlighted
User selects the target branch (e.g., main)	Target branch is set
User clicks Merge	Plugin sends merge request to backend
Backend computes differences between branches	Diff is calculated
No conflicts are detected	Merge proceeds automatically
Backend creates a merge commit on the target branch	Merge commit is recorded
Plugin notifies the user	Success message is displayed

Result : Final version of the program successfully passes this test

Test ID: F-TC-14

Test Type/Category: Functional, Integration

Summary / Title / Objective:
Verify that conflicting element changes between branches are detected and surfaced to the user during a merge .

Priority / Severity: Critical

Procedure of Testing Steps:

Action	Expected Result
Two users modify the same element on different branches	Diverging changes exist
User initiates merge of feature branch into main	Merge request is sent to backend
Backend detects conflicting element modifications	Conflict list is generated
Plugin receives conflict data	Merge is paused; conflicts are not auto-resolved
Conflict panel opens in the plugin	Conflicting elements are listed with details
User is prompted to resolve each conflict	Resolution options are presented

Result :Final version of the program successfully passes this test

Test ID: F-TC-15

Test Type/Category: Functional, Component

Summary / Title / Objective:
Verify that the Merge View correctly displays conflicting and non-conflicting changes side by side

Priority / Severity: Critical

Procedure of Testing Steps:

Action	Expected Result
User is in an active merge with detected conflicts	Conflict data is available
User opens the Merge View panel	Panel renders successfully
Panel displays source branch state	Source branch element data is shown
Panel displays target branch state	Target branch element data is shown
Conflicting elements are visually highlighted	Conflicts are distinguishable from clean changes
Non-conflicting changes are shown as auto-resolved	Accepted changes are marked separately

User selects a conflicting element	Detailed property-level differences are displayed

Result :Final version of the program successfully passes this test

Test ID: F-TC-16

Test Type/Category: Functional, Component

Summary / Title / Objective:
Verify that the user can resolve conflicts in Merge View by choosing a source or target version

Priority / Severity: Critical

Procedure of Testing Steps:

Action	**Expected Result**
Merge View is open with unresolved conflicts	Conflict list is visible
User selects a conflicting element	Resolution options appear (Accept Source / Accept Target)
User chooses Accept Source	Source branch value is marked as resolved
User selects another conflict and chooses Accept Target	Target branch value is applied
All conflicts are resolved	Complete Merge button becomes active
User clicks Complete Merge	Backend finalizes the merge commit
Plugin confirms success	Merge commit appears in target branch history

Result :Final version of the program successfully passes this test

Test ID: F-TC-17

Test Type/Category: Functional, Component

Summary / Title / Objective:
Verify that the Diff View accurately displays element-level changes between two selected commits or branches

Priority / Severity: Critical

Procedure of Testing Steps:

Action	Expected Result
User opens the Diff View panel	Panel renders successfully
User selects a base commit or branch	Base state is loaded
User selects a comparison commit or branch	Comparison state is loaded
Plugin requests diff from backend	Backend computes element differences
Diff results are returned	Panel displays added, removed, and modified elements
Added elements are visually distinct (e.g., green)	New elements are clearly marked
Removed elements are visually distinct (e.g., red)	Deleted elements are clearly marked
Modified elements show property-level changes	Changed properties are listed with before/after values

Result : Final version of the program successfully passes this test

Test ID: F-TC-18

Test Type/Category: Functional, Component

Summary / Title / Objective:
Verify that the Diff View supports filtering results by element type or change category

Priority / Severity: Minor

Procedure of Testing Steps:

Action	Expected Result
Diff View is open with results loaded	Full diff is displayed
User applies a filter for Added elements only	Only added elements remain visible
User applies a filter for a specific element category (e.g., Walls)	List narrows to that category
User clears all filters	Full diff results are restored

Result :Final version of the program successfully passes this test

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

Result : Final version of the program successfully passes this test

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

Result : Final version of the program successfully passes this test

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

Result : Final version of the program successfully passes this test

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

Result : Final version of the program successfully passes this test

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

Result : Final version of the program successfully passes this test

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

Result : Final version of the program successfully passes this test

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

Result : Final version of the program successfully passes this test

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

Result : Final version of the program successfully passes this test

Test ID: NF-TC-09

Test Type/Category: Performance

Summary / Title / Objective:
Verify that branch creation and switching complete within an acceptable time under normal conditions .

Priority / Severity: Major

Procedure of Testing Steps:

Action	Expected Result
User creates a new branch on a project with multiple commits	Branch is created in under 5 seconds
User switches between two branches	Model updates and branch switch completes in under 15 seconds
Operation completes without UI freeze	Plugin remains responsive throughout

Result :Final version of the program successfully passes this test

6 Maintenance Plan and Details

CollabHub is a university senior design project and is not currently deployed in a production environment. However, the following maintenance considerations have been identified for future development and potential real world deployment.

Bug Tracking and Issue Resolution: Any bugs or unexpected behaviors discovered during or after development are tracked through the team's GitHub repository using issues. Fixes are applied through pull requests reviewed by at least one other team member before being merged.

Dependency Management: The backend relies on Python packages managed through a requirements file, and the frontend relies on NuGet packages defined in the project file. These dependencies should be reviewed periodically for security updates, particularly the JWT and authentication-related libraries.

Revit API Compatibility: As Autodesk releases new versions of Revit annually, the plugin's API references and target framework may need to be updated to maintain compatibility. The current implementation targets Revit 2026 and .NET 8.

Database Migration: The current implementation uses in-memory storage suitable for development and testing. Transitioning to a persistent PostgreSQL instance for production use will require database migration scripts and schema versioning to be implemented.

Known Limitations: Certain Revit element types are not currently supported by the ElementApplier and are logged as unsupported during pull operations. These limitations are documented and can be addressed in future development cycles by extending the applier's element handling logic.

Documentation: The codebase includes inline comments and a QUICK_START.md file to assist future developers in understanding the system. This documentation should be kept up to date as the system evolves. Also the website presents some but limited information related to the project.

7 Other Project Elements
7.1 Consideration of Various Factors in Engineering Design
7.1.1 Constraints
7.1.1.1 Implementation Constraints
- CollabHub was developed primarily as an Autodesk Revit Plugin using C# and the .NET framework to ensure integration with the Revit environment.
- The backend was built using Python (specifically with FastAPI framework) as it is a framework that most of the team are experienced with.
- PostgreSQL is used as the relational database for storing metadata (commit logs, user info, branch pointers).
- The system complies 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]

7.1.1.2 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.

7.1.1.3 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. (e.g. 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.

7.1.1.4 Social Constraints
- Usability for Non-Programmers: Unlike software engineers, architects may not be familiar with usage of command line tools. The application provides 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.

7.1.2 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]

7.2 Ethics and Professional Responsibilities

The CollabHub team acknowledges several ethical obligations in developing a version control tool for the AEC industry.

Data Integrity and Public Safety: Since architectural and structural designs directly affect the safety of building occupants, the team is responsible for ensuring that CollabHub never silently corrupts or discards design data. Any operation resulting in data loss or conflict must be explicitly communicated to the user before being finalized.

Intellectual Property and Confidentiality: Design files contain proprietary and commercially sensitive information. The team is obligated to implement strict access control mechanisms ensuring project files are only accessible to authorized collaborators.

Transparency of Limitations: The team commits to honestly documenting known system limitations, including unsupported element types and conflicts requiring manual resolution, rather than misrepresenting the tool as a complete solution.

Adherence to Professional Standards: The development process follows established engineering standards including IEEE 830 and UML 2.5.1, reflecting the team's commitment to responsible engineering practice.

7.3 Teamwork Details

7.3.1 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 branching and merging also helped with push & pull functionalities.. İbrahim Çaycı handled project initialization, login, registration, authentication, and merge view. Ömer Edip Aras worked on testing, entities, and helped with backend related problems. Tuna Göksal handled database creation, contributed to merging operation and helped with resolving pulling issues.

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.

7.3.2 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.

.

7.3.3 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 took part in the backend and testing of the core functionalities and managed the last part of the project. 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.

,

7.3.4 Meeting Objectives

CollabHub has successfully met its core objectives as defined at the start of the project. The system delivers a functional Revit plugin that allows architects to initialize projects, push snapshots, pull changes, view version history, and perform branching operations directly within the Revit environment. The backend API supports commit management, branching, and differential change detection through the DiffEngine. The ElementApplier successfully reconstructs and applies remote changes to local Revit models. While some element types remain unsupported due to Revit API limitations,and merging of branches currently fails, the system fulfills its primary goal of providing a Git-inspired primitive version control workflow tailored specifically to BIM workflows in Autodesk Revit.

7.3.5 Problems Encountered and Our Solutions

During the project, we had stalled on one notably major problem. The Revit api did not provide native create functions, so the objects that we needed to use to implement the pull functionality were not being created successfully. Our team decided on to use a work-around, to copy the generic object templates, modifying their parameters and placing them at specific locations. This problem has been the most important and the biggest one throughout the project development, we were stuck on pull functionality longer than expected due to complexity of finding and implementing this solution. However we as a team successfully resolved the issue and completed pull functionality as well as the remaining features of the project. We also had a little trouble finding the convenient work load distribution among the team members as the project followed more of a rather sequential set of tasks that cannot be done in parallel with 5 team members at once. So we decided to split the tasks into subtasks that needs to be done by a group of 2-3 people at a specified timeframe, based on whoever is available. This approach facilitated the flow of the implementation and yielded productive outcomes for teammates.

7.4 New Knowledge Acquired and Applied

Developing CollabHub required the team to learn and apply knowledge significantly beyond standard coursework. The Autodesk Revit API introduced challenges around main-thread restrictions, requiring careful async task management to avoid UI deadlocks. The team gained practical experience with BIM data structures, learning how Revit internally represents building elements and how to extract, serialize, and reconstruct them programmatically. On the backend, the team deepened their understanding of delta-based storage and diffing algorithms for large structured datasets. Designing the branching model

introduced the team to commit graph structures similar to Git. Integrating WPF-based UI within a host application also presented unique challenges in window lifecycle and dockable panel management.

8 Conclusion and Future Work

CollabHub demonstrates that Git-inspired version control is a viable approach for managing BIM workflows in Autodesk Revit. By tracking element-level changes, supporting branching, and providing a visual diff and merge interface, the system addresses a genuine gap in the AEC industry's tooling landscape.

Future work includes expanding support for currently unsupported Revit element types, improving conflict resolution with a dedicated merge UI, and optimizing snapshot storage for large models. Deploying the backend to a cloud environment with persistent PostgreSQL storage would make CollabHub viable for real-world use beyond development. Extending support to other BIM platforms such as IFC-based tools would further broaden its applicability across the industry.

9 Glossary

- **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.

10 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

[7] 3D Repo Ltd., “3D Repo — BIM Collaboration Platform,” 2025. [Online]. Available:

https://3drepo.com/

[8] Autodesk, “Autodesk BIM Collaborate,” 2025. [Online]. Available:

https://www.autodesk.com/products/bim-collaborate/overview