

Offline-First Collaborative Code Editor with Intelligent Cloud Synchronization

Mr. K. Mahesh Raj¹, Desireddy Srujana², Asodu Spilojo³, Bontha Jacintha Ruth⁴, G.Sai Vivek⁵, Shiva Kumar⁶

¹Assistant Professor, Dept. of Computer Science and Engineering, HITAM, Telangana, India

²³⁴⁵⁶Student, Dept. of Computer Science and Engineering, HITAM, Telangana, India

Abstract - Web-based programming platforms make coding easier to access, but they usually depend on a stable Internet connection for loading projects and saving changes. This paper presents an Offline-First Collaborative Code Editor with Intelligent Cloud Synchronization, developed to keep coding work available during temporary network interruptions. The system stores project data and offline changes in IndexedDB and keeps pending operations in a local queue. When the connection returns, the synchronization engine sends the pending changes to the backend and updates the cloud-side project data. The platform also includes JWT-based authentication, project and file management, version history, conflict detection, and real-time collaboration through Socket.IO. The frontend uses React, Vite, Tailwind CSS, shadcn/ui, and Monaco Editor, while the backend uses Node.js, Express.js, and MongoDB. Functional testing showed that users could continue editing while offline and that queued changes could be processed after reconnection. The system provides a practical way to combine browser-based coding with local persistence and cloud synchronization.

Keywords: Offline-First, Collaborative Code Editor, IndexedDB, Cloud Synchronization, Synchronization Engine, Version History, Conflict Detection, Socket.IO.

1. INTRODUCTION

Web-based code editors have become useful for students and developers because they allow source code to be written and managed directly through a browser. They remove much of the setup needed in a traditional programming environment and make projects easier to access across devices. A common limitation, however, is the dependence on network connectivity for project access and cloud storage.

A temporary loss of connectivity can interrupt an otherwise normal coding session. A user may still be able to type code, but saving the latest changes to a remote server may not be possible. For students working with inconsistent networks, this can make regular practice and project work inconvenient.

The proposed Offline-First Collaborative Code Editor addresses this issue by treating local storage as an

important part of the application rather than as a temporary cache. Project information and code changes are stored in the browser using IndexedDB, while a synchronization engine keeps track of operations that still need to reach the backend.

Along with offline support, the system provides authentication, project management, code editing, version history, conflict detection, and collaborative communication. The implementation uses React and Monaco Editor on the client side and Node.js, Express.js, MongoDB, JWT, and Socket.IO on the server side.

1.1 Background and Offline-First Computing

Offline-first computing focuses on keeping useful application functions available when network access is interrupted. Instead of making every action depend on a remote server, the application keeps necessary information locally and synchronizes it later. In this project, IndexedDB stores project-related data and pending operations in the browser.

When connectivity becomes available again, the synchronization engine reads the pending operations and communicates with the backend. This model creates a clear path between offline work and cloud storage and reduces the chance that a temporary network failure will interrupt the user's coding session.

1.2 Need for Offline Collaborative Code Editing

A coding platform that is useful in real conditions should not stop being useful as soon as the network becomes unavailable. When the network connection is temporarily unavailable, users can continue editing locally available project files and performing supported project operations, while the changes are retained for later synchronization. The browser keeps these changes in local storage while the connection is unavailable. Once the network is restored, the queued operations are sent to the backend and the project state can be updated. Keeping the operations locally also makes it possible to track changes that are waiting for synchronization.

Collaboration also requires project members to share changes while working on the same project. The proposed

system therefore combines offline persistence with Socket.IO-based real-time communication when users are connected.

The system is designed as a browser-based workspace that brings code editing, project management, synchronization, and connected collaboration into the same application.

1.3 Motivation

The project was developed to reduce the dependence of the coding environment on a continuously available Internet connection. Students and developers may work in environments where connectivity is slow, intermittent, or temporarily unavailable. The intended behavior is to allow coding work to continue during a short network interruption without losing recently made changes.

1.4 Problem Statement

Many browser-based coding environments rely on a remote server to load project data and store changes. If that connection is lost, access to project data and saving recent code changes can be affected, and connected collaboration may also be interrupted. This project therefore focuses on building a browser-based programming environment that can support local coding activities during disconnection and synchronize the recorded changes after the connection returns.

1.5 Objectives

The project aims to provide browser-based code editing, offline access to programming projects, local storage of project changes through IndexedDB, queued operations for later synchronization, version history, conflict detection, authenticated project management, and collaboration for connected users.

1.6 Scope

The implemented scope includes the web interface, code and file management, browser-side persistence, offline operation queuing, cloud synchronization, version history, conflict detection, authentication, and Socket.IO-based collaboration for connected users. The current implementation does not claim CRDT-based conflict-free merging or a full compiler and multi-language execution environment; those capabilities remain possible extensions.

1.7 Contributions

The main contribution of the project is bringing offline code editing and cloud-based project management. The relevance to this project is the separation between local work and later synchronization. The proposed

together within a single browser application. The implementation brings together IndexedDB for local storage, an operation queue, a synchronization engine, version tracking, conflict handling, JWT authentication, and Socket.IO communication. Together, these components allow project work to continue locally when network availability changes and provide a path for synchronization after reconnection.

2. LITERATURE REVIEW

The reviewed literature covers local-first software, replicated data, browser storage, synchronization, and collaborative applications relevant to the proposed editor. These studies help explain how applications can retain local state during disconnection and synchronize that state with a remote system later.

Local-first research is particularly relevant because it changes the usual assumption that the server must be available before useful work can happen. The reviewed studies also show that synchronization becomes more complex when several copies of data can be changed independently.

2.1 Local-First and Offline-First Systems

Kuessner, Mogk, Wickert, and Mezini [1] studied Algebraic Replicated Data Types for secure local-first software. Their work considers replicated data, synchronization, end-to-end encryption, and conflict-free replicated structures. The study is relevant to the proposed system because it shows how replicated state can be handled when data is available at more than one location.

The main difference is that the proposed project is an end-user programming environment rather than a general replicated-data programming model. The ideas from this work are useful for understanding future improvements to synchronization and conflict handling.

The work also highlights an important point for offline systems: local changes cannot simply be discarded when the network disappears. They need a defined way to be retained and reconciled with other copies later.

2.2 Secure Replicated Data and Synchronization

Haas, Mogk, Yanakieva, Bieniusa, and Mezini [2] presented LoRe, a programming model for verifiably safe local-first software. The work focuses on local-first programming and consistency, with CRDT-related techniques for managing replicated state. It provides a useful conceptual foundation for systems in which local modifications are normal rather than exceptional.

editor follows the same broad local-first idea, but uses an operation queue and a backend synchronization process

rather than claiming a CRDT-based editor.

2.3 Local-First Collaborative 3D WebGIS Application

Wang et al. [3] described a Local-First Collaborative 3D WebGIS application using browser technologies including Progressive Web Applications, IndexedDB, CRDTs, and WebAssembly. The study demonstrates that a browser application can retain useful local state and synchronize distributed changes.

The domain is different from the proposed code editor, since WebGIS deals with geographic data rather than programming projects. Still, the use of IndexedDB is directly relevant because the same browser storage technology is used in the proposed system to retain project data and offline operations.

2.4 Local-First Programming Models

The reviewed studies address important parts of the offline-first problem, including replicated data, local-first programming models, and browser-based local storage. They do not, however, describe a single programming workspace that combines code editing, project management, authentication, IndexedDB persistence, queued synchronization, version history, conflict detection. The proposed work focuses on this practical integration. The main problem is therefore not only storing data offline, but connecting local editing, operation tracking, server synchronization, and project-level features in one browser application.

The identified gap also leaves room for future CRDT-based collaboration. In the current implementation, synchronization is handled through an explicit operation queue and conflict checks; CRDTs are considered as a future enhancement rather than as an implemented component.

The system is intended for situations where a user may move between connected and disconnected states during normal project work. This makes the transition between local and remote data an important part of the design. The literature therefore supports the use of local persistence and synchronization, while the proposed project applies these ideas specifically to a web-based programming environment.

2.5 Browser-Based Offline Applications

Existing online coding systems generally assume that the network is available when projects are loaded and changes are saved. When that assumption fails, the user can face delays, unavailable project data, or unsaved work.

A second issue is consistency. Once a local copy has changed, the system needs to know what should be sent to the server and how to react when the server-side state is different. A third issue is collaboration, where connected users may change shared project data around the same time.

The proposed project addresses these problems through four connected ideas: local persistence in IndexedDB, an operation queue for offline changes, a synchronization engine for reconnection, and mechanisms for version history and conflict detection.

2.6 Research Gap

The literature indicates that offline-first and local-first approaches can reduce dependence on continuous connectivity, but the reviewed work addresses synchronization mechanisms, programming models, or domain-specific applications separately. The identified gap is an integrated programming environment that applies these principles specifically to source-code projects and maintains a practical path between offline editing and server synchronization.

3. METHODOLOGY

The system is organized into a frontend layer, local storage layer, synchronization layer, and backend layer. The frontend provides the user interface and code editor. IndexedDB keeps project information and offline operations locally. The synchronization engine manages pending work, while the Node.js and Express.js backend communicates with MongoDB.

The architecture is designed so that local storage is not treated as an afterthought. It is part of the normal data path for project work. This allows the editor to read locally available information during offline periods and later pass pending changes to the server.

The main data flow can be summarized as follows:

Browser → React + Monaco Editor → IndexedDB → Offline Operation Queue → Sync Engine → Backend API → MongoDB

For connected collaboration, Socket.IO provides a real-time communication path between the client and server. This path is separate from the offline queue, since queued operations are concerned with changes that still need to be synchronized after a network interruption.

3.1 Research Method and System Design

IndexedDB is used to store project-related information in the browser. The project documentation identifies stores for files, folders, operations, and projects. This local data allows the application to retain useful project state even when the remote backend cannot be reached.

Keeping an operation record is important because the application needs to distinguish between the current local state and changes that have not yet been sent to the server.

3.2 Proposed System

Changes made while offline are placed in a local operation queue. Instead of treating an unavailable network as a failed save, the application records the operation and waits for connectivity to return.

The queue acts as a bridge between offline work and cloud synchronization. On reconnection, the synchronization engine checks the pending operations and processes them through the backend.

This design also makes synchronization easier to observe and debug because each pending operation has a defined place in the local data flow.

3.3 System Architecture

The synchronization engine is responsible for processing pending operations after connectivity is restored. It checks whether synchronization is already running, reads pending operations, processes them in sequence, and updates their status as the server accepts the corresponding changes.

The basic sequence is:

Offline edit → Store locally → Queue operation → Detect reconnection → Process pending operations → Update server → Refresh local state

This approach reduces the dependency on continuous connectivity and provides a controlled path for moving locally generated changes to the cloud.

3.4 System Workflow

A synchronization conflict can occur when the local and server-side states no longer represent the same version of a project or file. The implementation checks project state during synchronization and uses conflict-handling logic when inconsistent changes are detected.

Version history keeps previous project states so that project changes can be tracked over time. Together, version history and conflict handling provide a safer way to manage changes than simply overwriting the current state.

3.5 Authentication and Backend Communication

JWT-based authentication is used to control access to protected backend resources. After registration or login, the client uses the issued token when communicating with protected endpoints, while project operations are associated with authenticated users.

3.6 Synchronization State Management

The synchronization engine treats a local operation as pending until the backend confirms successful processing. This distinction prevents the application from treating an unsent operation as complete. Network availability is monitored so that synchronization can be triggered after reconnection, while repeated synchronization attempts are controlled to avoid overlapping processing.

3.7 Operation Processing Logic

For an offline change, the application first updates the locally available project state and records the corresponding operation. When synchronization is possible, the engine reads pending operations, processes them in the required sequence, communicates with the backend, and updates the local operation state after confirmation. If an operation cannot be completed, it remains available for a later attempt rather than being silently discarded.

3.8 Collaboration Model

The collaboration layer operates when users have network connectivity. Socket.IO provides the communication channel for connected project activity, while the offline-first layer remains responsible for local persistence during disconnection. This separation allows the system to support connected collaboration without making real-time communication a prerequisite for offline coding.

4. IMPLEMENTATION

The implementation uses a React-based frontend and a Node.js/Express.js backend. MongoDB stores persistent project information, while IndexedDB handles local project data and pending offline operations. The application also integrates Monaco Editor for source-code editing and Socket.IO for connected collaboration.

4.1 Frontend and Code Editor

The frontend was developed using React and Vite. Tailwind CSS and shadcn/ui are used for the interface, while Monaco Editor provides the main code-editing experience. The application includes login, registration, dashboard, project management, file management, and editor screens.

The editor is connected to the project's local and remote data flow. A user can open a project, select files, edit code, and continue working while the application is offline.

4.2 User Authentication and Project Management

JWT-based authentication controls access to protected resources. After registration or login, the client uses the issued token when communicating with protected backend endpoints.

Project management allows users to create and organize programming projects and their files. These operations are represented in the local data layer when offline so that project work does not depend completely on the server being reachable.

4.3 Offline Storage Implementation

The browser-side CodeSyncDB database stores files, folders, projects, and operations. When an offline change is made, the updated local state is retained and the corresponding operation is recorded for later synchronization.

4.4 Offline Operation Queue

The queue contains operations that have not yet been confirmed by the backend. The synchronization engine reads these entries after reconnection and sends them through the appropriate backend path. Completed operations can then be marked or removed according to the synchronization state.

The queue design is useful for operations such as creating, updating, or deleting project data because the action itself can be retained even when the network is temporarily unavailable.

4.5 Synchronization and Reconnection

The application monitors connectivity and starts synchronization when the network becomes available. The engine avoids overlapping synchronization runs, checks for pending work, processes the queue, and

updates local information after successful server communication.

This sequence keeps the offline workflow separate from normal online requests while still allowing the two modes to meet at the synchronization layer.

4.6 Version History

Version history records previous project states and gives the user a way to understand how a project has changed. This is useful when a recent edit needs to be reviewed or when synchronization produces a state that needs to be traced.

The version mechanism is implemented as part of the application rather than relying only on an external version-control system. It is intended to provide project-level history inside the editor.

4.7 Conflict Handling

During synchronization, local and server-side project states are compared so that inconsistent updates can be detected. The current implementation provides conflict-handling logic for these cases. It does not use CRDTs, so complex simultaneous character-level merging remains outside the current scope.

4.8 Real-Time Collaboration

Socket.IO provides the communication channel used for connected collaborative activity. This allows project users to exchange updates while online, while the offline storage and queue continue to protect local work when connectivity is unavailable.

4.9 Testing and Technology Stack

The implementation was tested in a local browser environment by checking the main user flows and by deliberately interrupting network connectivity. The tests focused on whether project work remained available offline and whether pending changes could be synchronized after reconnection.

4.10 Functional Test Cases

The following cases represent the main functions checked during testing: registration, login, project management, code editing, local persistence, synchronization, version history, conflict handling, and connected collaboration.

4.11 Technology Selection

The technologies were selected according to the role each

component plays in the system. React and Vite support the client interface, Monaco provides code editing, IndexedDB provides browser persistence, and Node.js/Express.js with MongoDB provide the server-side data path.

4.12 Offline Testing Procedure

For offline testing, network access was interrupted after the project was loaded. Code and project operations were then performed through the browser. The expected behavior was that the user could continue working and that the changes would remain in local storage rather than being lost.

After connectivity was restored, the synchronization process was observed for pending operations. The expected result was that queued changes would be sent to the backend and reflected in the server-side project state.

4.13 Collaboration Testing

Connected collaboration was checked through the Socket.IO communication layer. The purpose of this test was to verify that the collaboration mechanism could operate while users were online, while the offline-first mechanisms remained available as the fallback path during network interruption.

4.14 Editor Data Flow

When a user opens a project, the application loads the available project structure and files. Code edits update the editor state and are connected to the persistence and synchronization flow. Online changes can communicate with the backend, whereas offline changes are retained locally and associated with pending operations for later synchronization.

4.15 Local Database Structure

The browser-side CodeSyncDB database is used to maintain project-related information locally. The implementation uses stores associated with projects, files, folders, and operations. This structure allows the application to retrieve local project state during offline operation and gives the synchronization engine a persistent source of pending work after reconnection.

4.16 Reconnection Handling

The application monitors connectivity and starts synchronization when network access becomes available. The synchronization process checks whether another synchronization operation is already active,

reads pending operations, sends them through the backend path, and updates their status after processing. This controlled flow reduces the possibility of duplicate or overlapping synchronization attempts.

4.17 Security and Access Control

JWT-based authentication is used to control access to protected application resources. Authentication is handled through the frontend and backend so that project operations are associated with authenticated users. This provides the access-control foundation required for project management and connected collaboration.

5. RESULTS AND DISCUSSION

The functional tests showed that the major implemented modules behaved according to their intended flow. Registration and login provided authenticated access, projects could be managed, and code could be edited through the browser.

5.1 Offline Storage Result

During an interrupted network connection, project data and changes remained available through IndexedDB. The local operation queue retained pending work for later processing.

5.2 Synchronization Result

After reconnection, the synchronization engine detected pending operations and processed them through the backend. This provided the expected transition from local changes to server-side project data.

5.3 Version and Conflict Result

Version history maintained previous project states, while conflict-handling checks identified inconsistent local and server states during synchronization. These functions provided additional protection around project changes.

5.4 Collaboration Result

The Socket.IO layer provided a real-time communication path for connected collaborative activity. The result is a system that combines online collaboration with offline local persistence rather than treating the two modes as separate applications.

5.5 Functional Testing Summary

The overall test outcome was recorded as passed for the major functions implemented in the project. The testing was functional rather than a large-scale

benchmark. Therefore, the results demonstrate the behavior of the implemented features in the tested environment rather than performance across a large population of users.

The following table summarizes the main test cases.

- Registration - account creation and authenticated access
- Project Management - project and file operations
- Offline Editing - local changes retained during network loss
- Synchronization - queued operations processed after reconnection

The main result of the implementation is the integration of offline coding and cloud-based project management in one browser application. Instead of stopping project work when the network is unavailable, the application keeps local project state and records operations for later synchronization.

5.6 Offline-First Behavior

The offline tests demonstrated that the editor could continue to be used after network connectivity was interrupted. Changes were stored locally through IndexedDB, and pending operations were retained in the local queue.

This behavior is important because the useful part of an offline-first system is not simply having a local database. The local data must remain connected to the application's normal project workflow so that the user can continue working without manually switching to another tool.

5.7 Synchronization Behavior

The synchronization tests showed that pending work could be processed after connectivity returned. The synchronization engine provided a controlled path from queued local operations to backend updates.

The approach also makes the synchronization state visible to the application. If an operation is still pending, it remains in the queue rather than being treated as completed before the server confirms it.

5.8 Version History and Conflict Handling

Version history provides a record of earlier project states, while conflict detection checks for differences between local and server-side information. These features are useful because synchronization is not only a transfer problem; it is also a state-management problem.

The current conflict mechanism is intentionally simpler than CRDT-based merging. It identifies inconsistent states and applies the implemented handling logic, while more advanced conflict-free editing is left for future development.

5.9 Collaboration

Connected collaboration is supported through Socket.IO. This allows the application to exchange real-time project activity while users have network access. The offline-first layer remains responsible for protecting work during disconnection.

Together, these components give the platform two related modes: a connected mode for server communication and collaboration, and a disconnected mode centered on local persistence and queued synchronization.

Observed Offline Behavior

During functional testing, the application continued to provide access to locally available project information after network access was interrupted. Code and supported project operations could remain available locally, demonstrating the main continuity objective of the offline-first design.

Observed Synchronization Behavior

After connectivity was restored, the synchronization engine detected pending work and processed the queued operations through the backend.

Observed Version and Conflict Behavior

Version history provided previous project states that could be reviewed as project changes accumulated. Conflict detection compared local and server-side information when synchronization required it. The implemented mechanism is intentionally simpler than CRDT-based merging and is therefore presented as conflict detection and handling rather than conflict-free simultaneous editing.

Observed Collaboration Behavior

Connected collaboration was tested through the Socket.IO communication layer. The feature provides a real-time communication path for project activity when users are online, while offline work remains supported independently through local persistence and queued synchronization.

Testing Limitation

The evaluation was functional rather than a large-scale performance benchmark. The results demonstrate the behavior of the implemented features under the tested conditions, but they do not establish performance across large numbers of concurrent users, very large projects, or long-duration offline sessions.

Offline-First Collaborative Code Editor Architecture

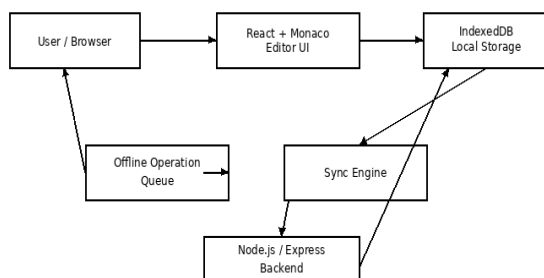


Fig. 1: Offline-First Collaborative Code Editor Architecture

Fig. 1 shows the main components involved in the proposed system. The browser communicates with the React and Monaco editor interface, which reads and writes project information through IndexedDB. Offline operations are retained in a queue and later passed to the synchronization engine and backend.

Component	Role
React + Monaco	User interface and code editing
IndexedDB	Local project and offline data
Operation Queue	Stores pending offline changes
Sync Engine	Processes queued changes after reconnection

Test	Expected Result
Registration/Login	Authenticated access
Project Management	Projects and files handled
Offline Editing	Changes retained locally
Synchronization	Pending operations processed

5.10 Discussion

The implementation shows that an offline-first approach can support a browser-based programming environment without removing the normal cloud-backed workflow. Local project state remains connected to the editor, while synchronization is handled separately.

When connectivity is lost, IndexedDB keeps local work available and the operation queue records changes that still need to reach the server. After reconnection, the synchronization engine processes the pending operations.

5.11 Practical Significance

For students and developers working with unstable connectivity, the system keeps project work available during temporary network failures.

Offline Editing and Cloud Synchronization Workflow

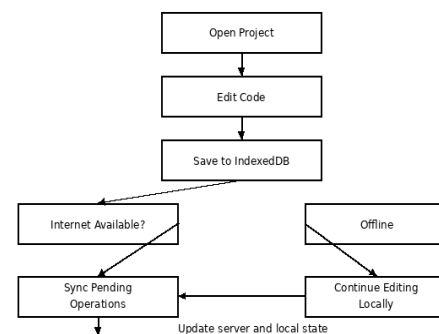


Fig. 2: Offline-to-Online Synchronization Workflow

5.12 System Limitations

The current implementation is intended as a functional prototype and was evaluated through functional testing rather than a large-scale benchmark. The system depends on browser storage limits for local persistence, and synchronization performance may vary with project size, network conditions, and the number of pending operations. The present conflict mechanism detects inconsistent local and server states and applies the implemented handling logic; it does not provide CRDT-based automatic merging of simultaneous edits. Real-time collaboration is supported for connected users through Socket.IO, while richer concurrent editing behavior remains an area for further development.

5.13 Overall Findings

The combined results show that the proposed architecture can maintain a useful coding workflow across changes in network availability. IndexedDB provides local persistence, the operation queue preserves work that still

needs server processing, and the synchronization engine provides a controlled path for reconnection. Version history adds traceability, while authentication and project management provide the surrounding application services. Together, these components demonstrate the feasibility of integrating offline-first behavior with a cloud-backed code editor without requiring the user to remain continuously connected.

6. CONCLUSION

The Offline-First Collaborative Code Editor with Intelligent Cloud Synchronization was designed to reduce interruption to coding when Internet connectivity is unavailable.

The system combines React, Monaco Editor, IndexedDB, an offline operation queue, a synchronization engine, Node.js, Express.js, MongoDB, JWT, version history, conflict detection, and Socket.IO collaboration.

Functional testing showed that users could continue editing offline, retain changes locally, and synchronize pending operations after reconnection.

The project provides a practical foundation for an offline-first programming environment; advanced replicated editing and large-scale evaluation remain future work.

6.1 FUTURE WORK

Future versions can investigate CRDT-based synchronization, richer collaboration, AI-assisted coding, multi-language execution, cloud deployment, and scalability testing.

CRDT-based synchronization can be investigated to support more advanced concurrent editing and reduce the need for application-level conflict handling when several replicas modify the same content. Any such extension would require careful evaluation of consistency, storage overhead, ordering, and merge behavior.

Real-time multi-user collaboration can be extended beyond the current connected communication layer by adding richer presence information, simultaneous editing support, and improved synchronization feedback. The editor can also be extended with code execution and testing services for multiple programming languages, subject to appropriate sandboxing and resource controls. AI-assisted features can be explored for code completion, explanations, error detection, and learning support. Cloud deployment and scalability experiments can evaluate the system with larger projects, longer offline periods, more queued operations, and larger numbers of

connected users. Version management can also be improved with detailed comparisons, restoration controls, and clearer change histories.

Version management can be improved with comparisons and restoration controls. Longer offline tests can study queue growth, retry behavior, synchronization ordering, and conflict reporting.

REFERENCES

- [1] Kuessner et al., "Algebraic Replicated Data Types: Programming Secure Local-First Software," ECOOP 2023. DOI: 10.4230/LIPIcs.ECOOP.2023.14.
- [2] J. Haas, R. Mogk, E. Yanakieva, A. Bieniusa, and M. Mezini, "LoRe: A Programming Model for Verifiably Safe Local-First Software," ACM Transactions on Programming Languages and Systems, vol. 46, no. 1, 2024, DOI: 10.1145/3633769.
- [3] B. Wang, Q. Zhao, D. Zeng, Y. Yao, C. Hu, and N. Luo, "Design and Development of a Local-First Collaborative 3D WebGIS Application for Mapping," ISPRS International Journal of Geo-Information, vol. 14, no. 4, Article 166, 2025, DOI: 10.3390/ijgi14040166.