A3: Architectural Enhancement Report
Group 13 – The Void Manual
Course: CISC 322/326 – Software Architecture

Abstract

This report proposes an architectural enhancement to the Void Editor: **native real-time collaborative editing**, enabling multiple developers to concurrently modify the same file with live cursor sharing, synchronized buffers, conflict-free merges, and session management. Extending the conceptual architecture from A1 and the recovered concrete architecture from A2, we examine how this feature integrates into Void's existing VS Code-based ecosystem, including the Editor Component, File Manager, Workspace services, and extension host mechanisms.

With this enhancement comes a new internal subsystem, which we'll call **Collaborative Service**, responsible for, among other things, session orchestration, network messaging, concurrency control, and conflict-resolution logic. It requires modifications to buffer-editing pathways and internal synchronization hooks. Two alternative realizations are evaluated using a **SAAM (Software Architecture Analysis Method)** framework, listed below:

1. **CRDT-based (Conflict-Free Replicated Data Type)** collaboration;
2. **Operational Transformation (OT)** collaboration.

Each alternative is assessed against stakeholder needs and architecturally significant non-functional requirements (NFRs) such as latency, correctness, scalability, maintainability, security, and failure tolerance. We further analyze how the proposed feature would affect subsystem dependencies, concurrency models, and team development workflows. The report includes architectural diagrams, use-case modelling, file and directory-level impacts, and test-planning considerations to conduct a full-picture analysis.

Our analysis concludes that **CRDTs** offer the more robust long-term architecture due to their support for offline editing, reduced coordination overhead, and deterministic conflict resolution, despite their higher memory and computational footprint.

1. Introduction & Overview

1.1 Purpose of the Report

The objective of this deliverable is to propose and evaluate an architectural enhancement to the Void Editor: **built-in real-time collaborative editing**. Building directly on A1's conceptual model and A2's concrete architecture recovered via SciTools Understand, this report examines how collaborative editing would reshape Void's subsystem structure and architectural relationships. In accordance with the established A3 requirements:

- Motivates the new feature and articulates its value to Void's stakeholders;
- Describes required modifications to the conceptual and concrete architectures;
- Analyzes impacts on fundamental NFRs;
    - Including performance, scalability, modifiability, testability, maintainability, reliability, and security;
- Presents two architectural alternatives and evaluates them through a structured **SAAM analysis**;
- Provides revised diagrams (functional, informational, and concurrency views);
- Documents file and subsystem-level impacts and risks;
- Includes our required AI-collaboration appendix.

This report is intended to allow a reader without prior exposure to A1 or A2 to understand both the rationale for the feature and its architectural implications on the entire system.

1.2 Enhancement Summary: Real-Time Collaborative Editing

Real-time collaboration enables multiple contributors to simultaneously view and modify the same file within Void, with synchronized cursors, live updates, merge-conflict resolution, and join/leave session support. Unlike Microsoft Live Share, which exists as an external extension, our feature is aimed at implementing collaboration inside Void's internal subsystems, modifying the core architecture rather than operating as a remote wrapper.

The proposed enhancement includes:

- A new **Collaboration Service** subsystem coordinating distributed editing sessions;
- Modifications to the **Editor Component**'s buffer-editing pathway to support synchronous updates;
- Low-latency messaging and merge logic (CRDT or OT);
- Integration with existing Void subsystems:
    - File Manager (buffer persistence and reload);
    - Workspace/Project Manager;
    - Security Layer (identity and permission checks);
    - Configuration System;

- Plugin Interface (to ensure compatibility with existing Void Extensions).

From an architectural perspective, this feature introduces new concurrency systems, new inter-subsystem connectors, and new NFR tradeoffs such as latency vs. correctness and performance vs. scalability, mirroring the tradeoff patterns emphasized in the course material and studied extensively throughout the term.

This feature is of **significant architectural complexity**, satisfying A3's requirement for substantial subsystem interaction.

1.3 Report Structure

Section	Focus
1	Introduction and overview of the proposed enhancement
2	Updated conceptual + concrete architecture models
3	Detailed description of the collaborative editing feature
4	Use cases and sequence diagrams
5	Architectural modifications and subsystem impacts
6	Two architectural alternatives (introduced above) + full SAAM analysis
7	Concurrency, project-team implications, risks, and limitations
8	Testing interactions between the new feature and existing Void functionality
9	Conclusions and recommended approach
Appendix	AI collaboration log and summary

Table 1: Report Structure Summary

A reader should be able to understand the feature motivation, the stakeholders involved, and the architectural consequences directly from this introduction and abstract.

2. Updated Architecture (Conceptual + Concrete Integration)

This section integrates the **A1 conceptual architecture** of the Void Editor with the **A2 concrete architecture** recovered from the VS Code host and the Void-related source files. Most importantly, it introduces the new **Collaboration Service** subsystem and shows how it fits into the existing layered and client–server style architecture.

2.1 Existing High-Level Architecture (from A1/A2)

In the concrete architecture from A2, the Void Editor operates as a VS Code extension inside the VS Code extension host. At a high level, the runtime environment can be divided into:

Top-Level:

- **VS Code Workbench** – Provides the UI shell, editor tabs, command palette, and main event loop that dispatches user actions (open file, type, save, run command) to extensions.
- **VS Code Platform Services** – Offer core services Void depends on:
  - **Files**: Workspace file system abstraction (open, read, write, watch)
  - **Storage:** Persistent storage for user/workspace settings and state
  - **Configuration:** Access to VS Code's settings model, configuration scopes, and schema.
  - **Extension API:** The extension-host API surface through which Void registers commands, contributes US, and listens to editor events.

Based on A2, the Void-specific behavior is primarily concentrated in the following files and folders:

- configurationEditingMain.ts : Central coordination point for configuration editing operations and integration with VS Code's configuration services.
- settingsDocumentHelper.ts: Helper for reading/writing settings documents and mapping them to VS Code's configuration model.
- importExportProfiles.ts: Supports import/export of configuration profiles (e.g., user settings presets).
- extensionsProposals.ts: Handles contributions/proposals from other extensions.
- Platform-specific folders (browser/, node/): Provide environment-specific implementations for browser vs. Node.js/desktop runtimes.

Collectively, these concrete modules iditing functionality described in A1.

mplement the **Void subsystem** and mediate between VS Code platform services and the higher-level e

Editor-Level (from A1 conceptual architecture):

- **Editor Component** (buffer, cursor, undo/redo)
- **Syntax & LSP Services**
- **File Manager + Persistence**
- **Plugin System**
- **Configuration System**
- **Security Layer**

2.2 New Subsystem: Collaboration Service

To support real-time collaborative editing, we introduce a new subsystem inside the Void architecture: the **Collaboration Service**. This subsystem is architecturally significant because it adds distributed state management, new connectors, and new quality-attribute considerations (latency, correctness, reliability).

Newly Introduced Collaboration Service:

Responsibilities:

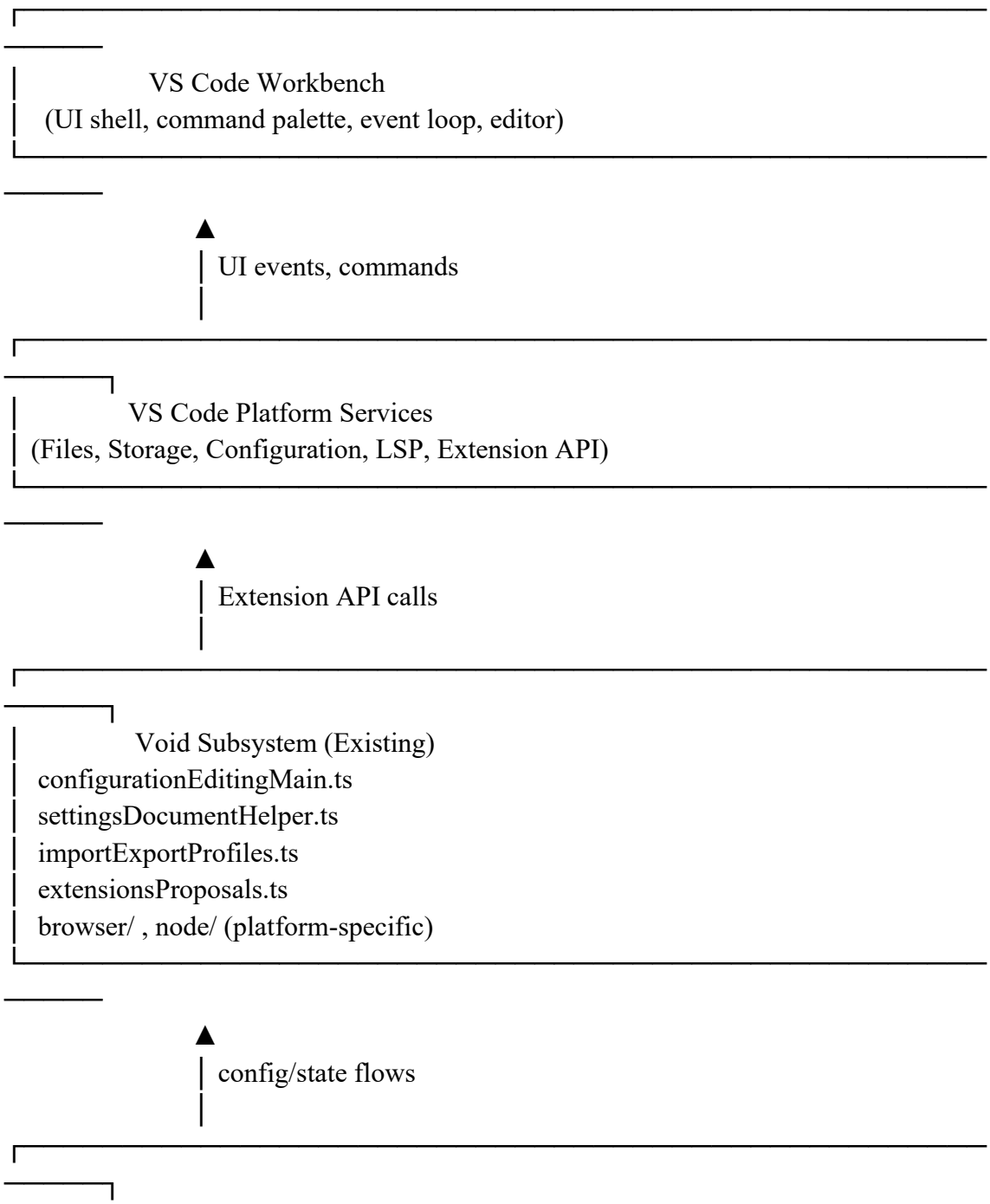
- **Collaborative session management:** create/join/leave sessions; maintain participant lists and session metadata.
- **Replicated document state:** maintain a CRDT/OT-based shared document model and ensure convergence across all participants.
- **Remote edit handling:** receive remote operations, transform/merge them, and update the local replica.
- **Local edit broadcasting:** intercept local edits from the Editor Component, convert them into operations, and broadcast them to collaborators.
- **Editor integration:** send normalized, conflict-free operations to the Editor Component so its buffer reflects the shared state.
- **Presence and cursor tracking:** manage and distribute remote cursor and selection metadata.
- **Network communication:** exchange operations through WebSockets or VS Code remote APIs.

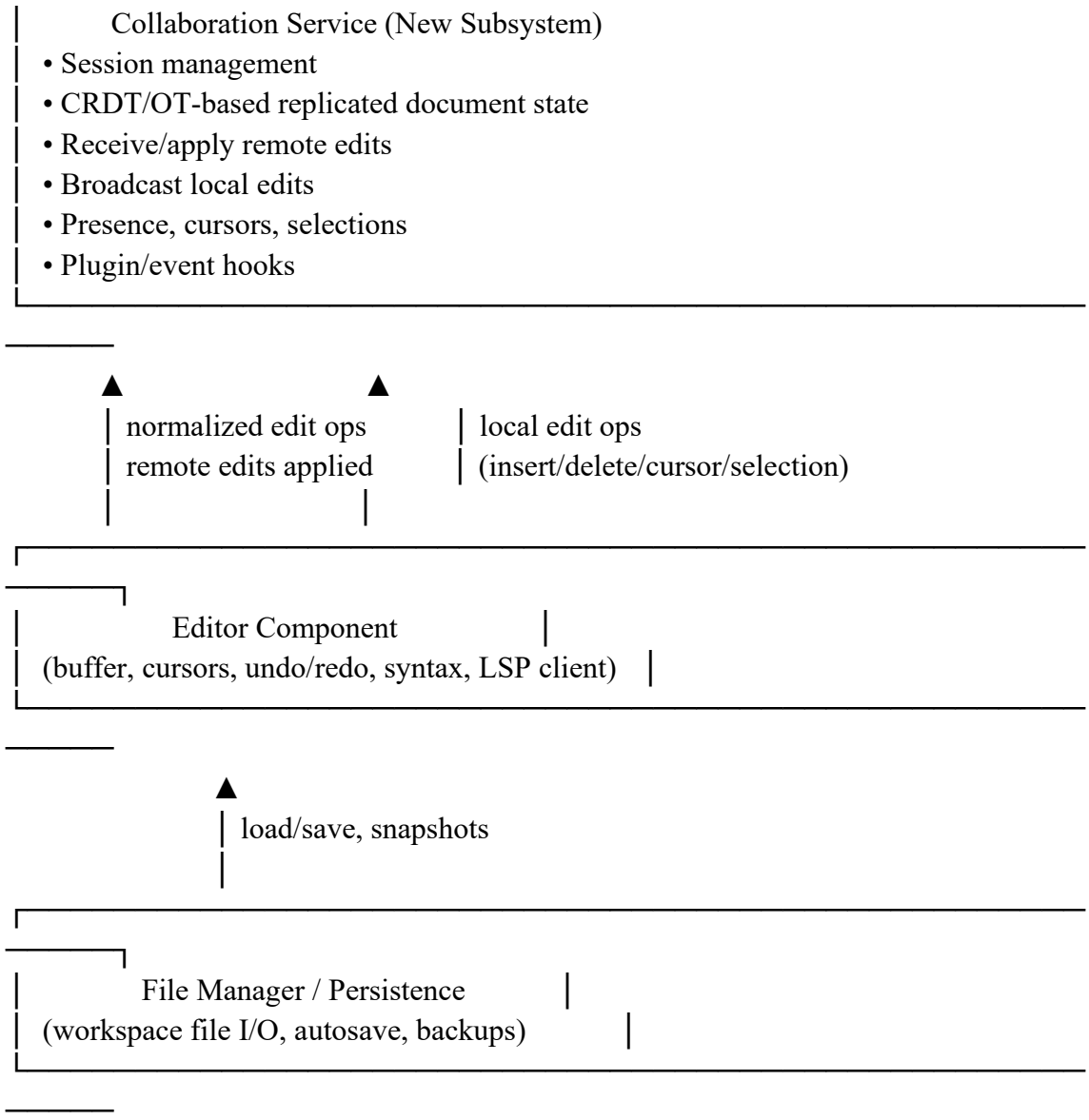
Interfaces

- **Incoming operations from Editor Component:**
  {insert, delete, cursorMove, selectionChange}
  converted into CRDT/OT operations for the shared state.

- **Outgoing operations to Editor Component:**
  normalized remote and transformed local operations applied to the buffer.
- **Plugin hooks:**
  events such as onRemoteEdit, onPresenceUpdate, or onSessionJoin for integration with the existing Plugin System.

2.3 Updated Architectural Diagram





3. Enhancement Description: Real-Time Collaborative Editing

3.1 Goals & Motivation

The primary goal of this enhancement is to enable developers to work together in the Void Editor as if they were sharing the same local environment. Modern development teams frequently operate across time zones and remote environments, making synchronous editing an increasingly essential capability. By embedding collaboration directly into Void's architecture, this feature supports workflows such as pair

programming, mentoring, collaborative debugging, and instructional demonstrations without relying on external extensions.
The architectural motivation for implementing collaboration as a native subsystem rather than a plugin includes:

- Low-latency, fine-grained synchronization integrated directly with the Editor Component's buffer model.
- Consistent editing semantics, ensuring cursor movement, selections, and undo/redo behave predictably in shared sessions.
- Cross-subsystem integration, allowing Configuration, File Manager, and Security Layer features to remain coherent in collaborative mode.
- Improved resilience, with the ability to tolerate intermittent connectivity and maintain document consistency.
- Controlled, predictable behaviour, since collaboration is implemented inside the Void subsystem rather than offloading logic to the VS Code extension ecosystem.

Overall, the enhancement expands Void's capabilities to support distributed development natively while ensuring alignment with its existing architectural principles.

3.2 Key Features

The collaborative editing feature introduces several user-visible and architectural capabilities:

- **Concurrent Editing**: Multiple participants can edit the same file simultaneously, with updates reflected across all instances in near real time.
- **Conflict-Free Synchronization**: All document replicas converge using a deterministic transformation model (CRDT or OT).
- **Shared Presence Metadata**:
    - Remote cursors and selections
    - User presence indicators
    - Participant list and editing focus
- **Session Lifecycle Management**: Users may create, join, leave, and resume collaboration sessions. The subsystem manages permissions, session discovery, and document retrieval.
- **Offline Tolerance (for CRDT-based realization):** Temporary disconnections do not prevent progress; updates are eventually synchronized.
- **Integrated Security Controls**: The Security Layer validates invitations, enforces permissions, and protects collaborative session data.
- **Subsystem Interoperability**: The feature integrates cleanly with:
    - Editor Component (buffer + cursor updates)
    - File Manager (consistent saving during collaboration)
    - Configuration System (collaboration preferences)
    - LSP/Syntax services (incremental reanalysis after remote edits)

3.3 How the Feature Operates Internally

To support collaboration, the enhancement introduces a new Collaboration Service subsystem responsible for coordinating all shared editing activity. Internally, it:

- Maintains a synchronized representation of the shared document.
- Encodes outgoing edits as operations and broadcasts them to other participants.
- Receives remote operations and applies them to the local replica using CRDT/OT logic.
- Tracks and distributes metadata such as cursors, selections, and presence indicators.
- Manages the networking layer (e.g., WebSocket channels or host-provided messaging APIs).
- Coordinates the session lifecycle, including state initialization and reconnection.

The Editor Component routes all collaborative edit-related activity through the Collaboration Service, ensuring that both local and remote operations are applied through a consistent, correctness-preserving pathway.

3.4 Architectural Rationale

Implementing collaboration as a first-class subsystem (rather than an extension-level feature) ensures:

- Consistent integration with Void's internal buffer model
- Predictable undo/redo behaviour per user
- Reduced architectural drift across future features that depend on shared state
- Clear, modular separation of concerns between editing logic and synchronization logic

This structure also lays the groundwork for additional future enhancements such as shared terminals, collaborative debugging, and multi-user annotations.

4. Use Cases

4.1 Use Case 1 — Join a Collaboration Session

Actor: User
Preconditions:

- Void Editor running
- The session host has created a collaboration session

Flow:

1. User enters the invitation code / URL.
2. The Collaboration Service requests session metadata from the host.
3. Host validates permissions via the Security Layer.
4. Collaboration Service loads the shared document state.
5. Editor Component initializes the shared buffer.
6. UI Layer displays remote cursors.

Postconditions:

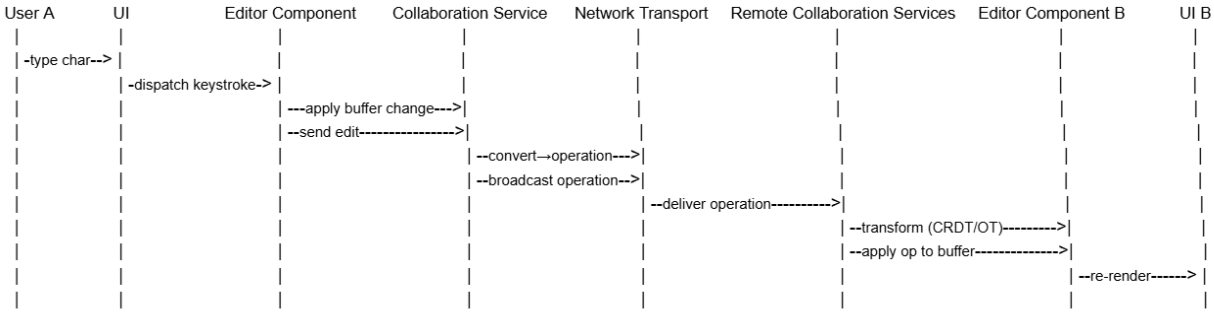
- User is fully synced with the session.
- Background workers maintain real-time connectivity.

4.2 Use Case 2 — Concurrent Editing (incl. Sequence Diagram)

Goal

Two users edit the same file at the same time; both see each other's updates instantly.

Placeholder Sequence Diagram



User A -> UI
UI -> Editor Component
Editor Component -> Collaboration Service
Collaboration Service -> Network Transport
Network Transport -> Remote Collaboration Services (User B)
Remote Collaboration Services -> Editor Component B
Editor Component B -> UI B

Flow Description

1. User A types a character.
2. UI dispatches keystroke → Editor Component.

3. Editor Component applies a change to the buffer.
4. Collaboration Service converts edit → operation.
5. Collaboration Service broadcasts operation.
6. User B receives the operation.
7. Collaboration Service on User B applies transformation (CRDT/OT).
8. Editor Component B updates the buffer.
9. UI B re-renders.

5. Architectural Modifications

Real-time collaborative editing is not a "local" change: it affects how the editor buffer is owned, how files are persisted, how permissions are enforced, and how UI state is synchronized across multiple users. This section ties those conceptual changes (Editor Component, Collaboration Service, File Manager, Configuration, Security, Persistence, UI) to concrete source directories and files in the Void / VS Code codebase, as required for A3.

5.1 Required Subsystem Changes

The table below summarizes, for each affected subsystem, (i) what changes are required, and (ii) why those changes are architecturally necessary for real-time collaboration and the relevant quality attributes.

Subsystem	**Change Required**	**Rationale / Architectural Impact**
Editor Component	Extend the editor's buffer API to accept remote operations and metadata (e.g., {insert, delete, cursorMove, selectionChange}), and to tag edits with session/user identifiers.	The Editor Component remains the single source of truth for the local buffer, but it must now distinguish between local keystrokes and remote CRDT/OT operations. This preserves existing undo/redo behaviour while enabling low-latency application of remote edits. It directly impacts the performance, correctness, and testability of editing behaviour under concurrency.

Collaboration Service (new)	Introduce a new Collaboration Service subsystem that owns the real-time session logic: CRDT/OT engine, session life cycle, network transport, and user presence (cursors, selections).	Architecturally, this inserts a clear collaboration layer between the Editor Component and any network/remote infrastructure. It centralizes concurrency control and protocol handling instead of scattering it across UI or file code. This improves modifiability (we can swap CRDT vs OT), maintainability, and reliability (single place to reason about convergence and failure modes).
File Manager	Update file open/save logic to avoid overwrite conflicts for shared files (e.g., version vectors or session IDs), and to be aware of "collaborative sessions" when writing to disk.	When a file is part of an active collaboration session, the File Manager must treat the in-memory document as a shared replica rather than a purely local buffer. This prevents lost updates on save and ensures persistence remains consistent with the collaboration model. This primarily affects reliability and data integrity, and slightly performance (extra checks on save).
Persistence Layer	Add support for collaborative draft checkpoints (e.g., periodic snapshots plus operation logs) and optional recovery of prior collaborative states.	Collaborative sessions generate many fine-grained operations; persisting only full document snapshots is inefficient and risky. Checkpoints plus logs improve failure resilience (crash recovery), support debugging of desynchronization bugs, and help performance testing (we can replay sessions).
Configuration System	Extend configuration to include collaboration-related settings: enabling/disabling collaboration, maximum participants, CRDT vs OT selection, security policies (e.g., "only	These settings map stakeholder preferences into runtime behaviour and make the enhancement evolvable without code changes. They also bridge conceptual architecture (Collaboration Service & Security Layer policies) with concrete configuration files and UI controls.

	workspace owner can host sessions").	
Security Layer	Introduce session-level access control: invitation tokens or session IDs, role distinctions (host vs participant), and validation hooks before applying remote operations.	Real-time collaboration amplifies security risks (untrusted remote edits, unauthorized joins). By routing all join/operation requests through the Security Layer, we preserve the existing security architecture and keep authorization concerns separate from editing logic. This improves security, auditability, and maintainability.
UI Layer	Extend the UI to display participant lists, colored cursors, remote selections, and connection status; add entry points for "Start Collaboration Session" and "Join Session".	The UI surfaces the collaboration state to users but should delegate logic to the Collaboration Service and Editor Component. The impact is mainly on usability and learnability, but also on performance (extra rendering) and testability (UI behaviour under rapid updates).
Network / Transport (within Collaboration Service)	Define a collaboration-specific messaging contract (e.g., over WebSocket or VS Code remote APIs) for broadcasting operations and presence updates, with reconnection and backoff.	While not a separate conceptual subsystem in A1/A2, network transport is now a first-class concern for the Collaboration Service. Clean message boundaries and error handling improve reliability, latency, and testability (we can simulate network failures and delayed messages).

Table 2: Required Subsystem Changes

Overall, these changes preserve the original high-level structure (Void inside VS Code, Editor Component owning the buffer) while inserting the Collaboration Service as a new subsystem that coordinates concurrency and session semantics.

5.2 Impacted Files & Directories

This subsection links the conceptual subsystem changes above to **concrete files and directories** in the VS Code + Void codebase, as expected in the A3 rubric.

5.2.1 VS Code / Platform Directories (High-Level)

These directories correspond to platform-level services that Void already depends on; collaborative editing extends them but does not radically change their responsibilities.

- **src/vs/editor/*** — *Editor Component & UI Layer*
  - **Why impacted**:
    - Integration of new APIs for applying remote operations and cursor metadata.
    - Rendering of multi-user cursors, selections, and presence indicators.
  - **Examples of concrete changes**:
    - New interfaces or methods in shared editor model files (e.g., edit application methods that accept collaboration metadata).
    - UI components updated to render participant cursors and connection state.

- **src/vs/platform/files/*** — *File Manager*
  - **Why impacted**:
    - Save/open paths now need to be collaboration-aware and avoid overwriting shared buffers.
  - **Concrete changes**:
    - Additional file versioning or “collaborative session” flags in file service APIs.
    - Hooks to notify the Collaboration Service when files are saved or closed, so sessions can be terminated safely.

- **src/vs/platform/storage/*** — *Persistence Layer*
  - **Why impacted**:
    - Storage of collaboration checkpoints, session metadata, and possibly operation logs.

    - **Concrete changes**:
        - New storage keys/namespaces for collaborative drafts.
        - APIs for serializing/deserializing CRDT structures or operation logs.

- **src/vs/platform/configuration/*** — *Configuration System*
    - **Why impacted**:
        - Addition of collaboration settings (e.g., feature toggle, CRDT vs OT, security policies).
    - **Concrete changes**:
        - New configuration schema entries for collaboration.
        - Validation logic for collaboration-related settings and exposure to the Void UI.

- **src/vs/platform/security/*** — *Security Layer*
    - **Why impacted**:
        - Enforcement of session permissions and validation of join/invite tokens.
    - **Concrete changes**:
        - New security checks and policies for collaboration actions (create session, join session, apply operation).
        - Possible integration with existing authentication/authorization mechanisms (e.g., reuse of existing principals).

These directories are mostly part of the underlying VS Code platform; the collaboration enhancement extends them in a way that's consistent with their existing responsibilities rather than introducing new cross-cutting hacks.

5.2.2 Void-Specific Files & Directories

These files and directories belong to the Void extension itself and therefore carry most of the Void-specific collaboration behaviour.

- **void/configurationEditingMain.ts**
    - **Role**: Coordination entry point for configuration editing within Void.
    - **Collaboration impact**:
        - Registers new collaboration-related configuration options.

        - Wires configuration changes into the Collaboration Service (e.g., enabling/disabling collaboration, selecting CRDT vs OT).

- **void/settingsDocumentHelper.ts**
    - **Role**: Helper for reading/writing settings and configuration documents.
    - **Collaboration impact**:
        - Ensures configuration changes made in a session are correctly propagated to all participants.
        - May provide utilities for resolving setting conflicts in collaborative scenarios.

- **void/browser/**
    - **Role**: Browser-side Void components, including views and UI logic.
    - **Collaboration impact**:
        - New views for starting/joining collaboration sessions.
        - Components for rendering participant lists, colored cursors, and connection status.
        - Event wiring between UI controls and the Collaboration Service API.

- **void/node/**
    - **Role**: Node-side logic for Void (server-like/backend behaviour).
    - **Collaboration impact**:
        - Optional hosting of a lightweight coordination or relay service when running Void with a backend.
        - Integration with VS Code remote APIs or other transport mechanisms used by the Collaboration Service.

These modifications keep collaboration logic inside the Void subsystem while reusing the shared VS Code platform for files, storage, configuration, and security.

5.2.3 New Directories and Files

To avoid scattering collaboration logic across unrelated modules, we introduce a dedicated collaboration package within the Void subsystem:

- **New directory:** void/collaboration/
  - **collaborationService.ts** – *Subsystem façade*
    - Exposes a clear API to the rest of Void: start/end session, join session, broadcast local edits, apply remote edits, and manage user presence.
    - Internally coordinates CRDT/OT logic, network transport, and error handling.
    - Acts as the main realization of the **Collaboration Service** subsystem in the conceptual architecture.
  - **operations.ts** – *Shared operation model*
    - Defines the canonical operation types used by collaboration (insert, delete, cursor move, selection change, etc.) plus any CRDT/OT metadata.
    - Centralizing these types improves **maintainability** and **testability**, and allows the same operation model to be reused by both the Editor Component and the network transport.
  - **sessionManager.ts** – *Session life-cycle management*
    - Manages creation, joining, and termination of collaboration sessions, including mapping from files to active sessions.
    - Handles reconnection logic, timeouts, and cleanup when hosts or participants disconnect.
    - Provides hooks to the Security Layer to validate invitations and permissions.

By grouping these new files under void/collaboration/, the architecture maintains a clean mapping from the conceptual Collaboration Service to a cohesive concrete implementation. This organization supports the report's goals of **clear traceability** between conceptual subsystems and code-level structure, and makes future changes (e.g., switching from OT to CRDT or vice versa) easier to implement in a single part of the codebase.

5.3 Summary of Architectural Impact on Quality Attributes

The introduction of a dedicated Collaboration Service and related changes primarily target **modifiability** and **maintainability**. By centralizing real-time session logic, concurrency control, and operation handling under void/collaboration/, we avoid

scattering collaboration concerns throughout the editor and file subsystems. This clean separation makes it easier to evolve the synchronization strategy (e.g., switching between OT and CRDT) and to extend collaboration features without destabilizing core editing behaviour.

The enhancement also significantly affects **reliability**, **consistency**, and **performance**. Collaboration-aware file saving and persistence (checkpoints plus operation logs) reduce the risk of lost updates and provide a structured way to recover from crashes or desynchronization. At the same time, keeping the Editor Component as the single source of truth for the local buffer and pushing only well-structured operations through the Collaboration Service helps maintain good runtime performance and predictable convergence under concurrent edits.

Security and user experience are handled as first-class concerns rather than afterthoughts. The integration with the **Security Layer** (session permissions, access control) ensures that only authorized users can join sessions or apply remote edits, improving the **security** and **trustworthiness** of the system. The **UI Layer** changes (participant list, colored cursors, connection indicators) directly support **usability** and **learnability**, making the collaborative behaviour visible and understandable. Finally, the use of explicit operation models and a clear module boundary around collaboration improves **testability**: collaboration logic can be unit-tested with synthetic operation streams and fault-injected network scenarios, rather than only through end-to-end manual testing.

6. Alternatives & SAAM Analysis

6.1 Alternative 1 — CRDT-Based Collaboration

In the first alternative, VOID implements collaboration using Conflict-free Replicated Data Types (CRDTs). Each client maintains its own local replica of the document. Edits are applied immediately and propagated asynchronously to other collaborators. CRDTs ensure strong eventual consistency, meaning all replicas converge even in the presence of concurrent or out-of-order operations.

Pros:

- **No central coordination server required**; correctness is guaranteed through mathematical convergence properties.
- **Strong eventual consistency**, even under high concurrency.
- **Offline-first model**, allowing edits to be made without network connectivity and synchronized later.
- **High reliability** during network partitions or intermittent connectivity.

Cons:

- **Higher memory usage** due to per-operation metadata such as unique identifiers or tombstones.
- **Larger operation payloads**, which can increase bandwidth usage under heavy collaboration sessions.

Overall, CRDTs emphasize robustness, decentralization, and offline tolerance.

6.2 Alternative 2 — Operational Transformation (OT)

The second alternative uses **Operational Transformation (OT)**, a widely deployed approach in systems such as Google Docs. Clients send operations (insert, delete, replace) to a **coordination server**, which transforms incoming operations based on the global operation history to preserve user intent. The transformed operations are then broadcast back to all participants.

Pros:

- **Highly efficient** when many users edit concurrently.
- **Low memory footprint**, as operations do not require substantial metadata.
- **Mature approach** with extensive academic and industrial adoption.

Cons:

- **Requires a central coordination server**, introducing a single point of failure.
- **Complex transformation logic** is more difficult to implement and maintain correctly.
- **Not offline-first**, as offline edits require non-trivial reconciliation.

OT optimizes runtime efficiency at the cost of higher architectural complexity and reduced offline capability.

6.3 Stakeholders

The collaborative editing enhancement affects multiple stakeholder groups:

Developers (End Users) ： require low latency, correctness, and fluid collaborative workflows.

Team leads / managers ： value reliability, consistency, and stable coordination across teams.

Instructors / mentors ： depend on predictable, real-time collaboration for demonstrations and code reviews.

Plugin developers ： need stable APIs and easily extensible collaboration mechanisms.

System integrators ： require clear interfaces, secure communication, and good interoperability.

Performance testers ： evaluate responsiveness, concurrency behaviour, and robustness under load.

DevOps / maintainers ： manage deployment, monitoring, and operational stability (especially for OT-based systems).

Different stakeholders emphasize different NFRs, forming the basis for architectural evaluation.

6.4 NFRs (Non-Functional Requirements)

The following NFRs were identified as most relevant to evaluating the two alternatives:

Latency： The speed at which remote edits propagate.

Scalability： Ability to support large numbers of collaborators and large documents.

Offline tolerance： Ability to operate without continuous connectivity.

Maintainability： Ease of debugging, understanding, and evolving the subsystem.

Modifiability： Flexibility to incorporate new features or protocol extensions.

Reliability： Ability to maintain correctness and convergence under failures.

Security： Protection of session data through proper authentication and authorization.

These NFRs guide the architecture-level comparison.

6.5 SAAM Evaluation Table

A qualitative scoring system (1–10) is used to compare how well each alternative satisfies each NFR.

NFR	CRDT Score	OT Score	Notes
Latency	8	9	OT operations are smaller and propagate faster.
Scalability	9	7	CRDT avoids central server bottlenecks.
Maintainability	8	6	OT transformation is more complex.
Modifiability	9	6	CRDT's model is easier to extend.
Offline Support	10	4	CRDT is inherently offline-first.
Security	8	8	Security depends mostly on session authentication.
Reliability	9	7	CRDT ensures reliable convergence under failures.

Table 3: Comparison between CRDT and OT

The table demonstrates that CRDT outperforms OT in nearly all NFRs, except for latency and operation size, where OT is slightly better.

Conclusion:

Based on our comparison, the CRDT approach better fits the needs of the Void Editor. It works well even when the network is unstable, lets people keep editing offline, and naturally scales without depending on a central server. These strengths match what most of our users and teams would expect from collaborative editing.

OT definitely has its benefits, mainly its mature ecosystem and slightly lower latency. But the need for a coordination server and its weaker support for offline work make it a less ideal long-term choice for VOID's architecture.

So overall, CRDT is the option we recommend, as it offers a more reliable and flexible foundation for the collaboration feature we want to build.

7. Risks, Interactions & Impact

7.1 Risks

Integrating real-time collaboration introduces several architectural risks that must be managed:

- **Network Instability**: Loss of connectivity may interrupt synchronization. In CRDT-based realizations, this is mitigated by offline-first operation and eventual convergence.
- **Unauthorized Access to Sessions**: Exposed collaboration endpoints could allow unintended users to join sessions. This risk is mitigated through the Security Layer's permission checks, authenticated invitations, and controlled session initiation.
- **Performance Degradation Under High Activity**: Frequent remote operations or large shared documents may increase CPU and memory usage. Throttling, batching, and incremental updates help maintain responsiveness.
- **Editor Consistency Errors**: Incorrect application of remote edits could destabilize the buffer state. Buffering and validation layers in the Collaboration Service ensure operations are applied safely.
- **Data Loss During Concurrent Saves**: Multiple users saving simultaneously can create conflicts. The File Manager incorporates conflict avoidance rules and save coordination mechanisms.

7.2 Interactions with Other Features

The collaborative editing enhancement interacts with several existing Void subsystems. Each interaction introduces architectural considerations:

Feature	Interaction
Syntax Highlighting	Tokenization must recompute incrementally after remote edits to prevent invalid states.
LSP Services	Multi-user diagnostics may produce overlapping or conflicting updates, requiring coordination to present unified feedback.
File Manager	Save operations must account for concurrent edits and ensure consistent file-state practice.
Plugin System	Plugins may receive new collaboration-related events, requiring stable APIs and compatibility guarantees.
Undo/Redo Stack	Each participant maintains an independent undo/redo history; operations must not interfere with remote users' histories.

Table 4: Implications on other Void features

These interactions collectively influence maintainability, modifiability, and reliability across the Void architecture. The Collaboration Service must ensure that shared editing behaviour integrates smoothly with each subsystem without violating existing invariants.

8. Conclusions & Lessons Learned

8.1 Conclusions

The addition of native real-time collaborative editing represents a substantial architectural enhancement to the Void Editor, requiring coordinated modifications across the Editor Component, File Manager, Configuration System, Security Layer, and a newly introduced Collaboration Service subsystem. Our analysis shows that the feature integrates naturally into the existing architecture while preserving the system's foundational principles: a centralized Editor Component that owns buffer state, clear subsystem boundaries, and a clean layering between UI, platform services, and editor logic.

After evaluating two viable architectural realizations, Operational Transformation (OT) and Conflict-Free Replicated Data Types (CRDTs), the team concludes that CRDT-based collaboration offers the stronger long-term fit for Void. CRDTs align well with Void's offline-friendly design, minimize reliance on a central authority, support deterministic convergence under concurrent edits, and simplify future extensibility. Although OT provides slightly lower latency, its dependence on a coordination server and reduced offline tolerance make it less compatible with Void's architecture and target use cases.

Overall, real-time collaborative editing significantly improves Void's functionality and positions it as a more powerful tool for distributed development, mentoring, and code review workflows. The enhancement is architecturally viable, preserves system integrity, and meaningfully expands the editor's capability set.

8.2 Lessons Learned

Through the design and evaluation process, we identified several key architectural insights:

- **Buffer ownership must remain centralized.** Even in a distributed editing environment, the Editor Component must continue to act as the authoritative owner of local buffer state

to maintain consistency and predictable undo/redo behavior.

- **Conflict-resolution models have system-wide consequences.** CRDT and OT differ not only in performance but also in how they influence the structure of buffer APIs, save logic, session management, and concurrency handling.

- **Deep integration with the VS Code platform is unavoidable.** File services, configuration, storage, and security mechanisms must all be extended or adapted to support collaborative flows without violating existing invariants.

- **Architectural drift requires constant vigilance.** Enhancements like collaboration challenge the boundaries between conceptual and concrete architecture; maintaining clarity around responsibilities prevents unintended coupling.

- **Offline-first architectures benefit from CRDT-like models.** CRDTs simplify recovery, reduce the impact of temporary disconnections, and provide more predictable behavior across varied network conditions.

These lessons reinforce the importance of modularity, well-defined subsystem boundaries, and rigorous architectural evaluation when integrating complex new functionality into an established system.

Appendix A — AI Collaboration Report

A.1 AI Member Profile

Our group's virtual AI teammate for A3 was **OpenAI GPT-5** (ChatGPT Plus). GPT-5 was used due to its strong ability to improve clarity, structure, and writing consistency across long technical documents. As in A1 and A2, all architectural decisions, subsystem mapping, SAAM evaluation, and code-level analysis were performed entirely by human members; the AI's role was limited to writing support.

A.2 Tasks Assigned to the AI

The AI was used selectively for low-risk, non-analytical tasks:

- Polishing grammar, tone, and flow of human-written paragraphs

- Standardizing terminology across sections written by different group members

- Suggesting concise wording for transitions and section openings
- Helping reformat tables for readability
- Creating short bullet-point summaries when requested

All technically meaningful content (use cases, SAAM scoring, subsystem interactions, architecture diagrams, file-level analysis) was fully human-generated.

A.3 Interaction Protocol

One group member acted as the primary point of contact, providing the AI with:

1. Human-written draft paragraphs
2. Clear instructions to “rewrite for clarity” or “shorten without losing meaning”
3. Constraints prohibiting technical reinterpretation

Each AI-generated revision was reviewed by at least one additional team member to confirm that:

- No technical meaning was altered
- No new facts were introduced
- Tone and structure remained appropriate for academic work

This workflow mirrors the approach established in A1 and A2.

A.4 Validation & Quality Control

To preserve academic integrity:

- AI outputs were considered *suggestions*, not authoritative content

- Each AI-edited section was cross-checked against A2's concrete architecture findings
- No diagrams, models, architectural mappings, or SAAM evaluations were generated by AI
- Human contributors finalized all technical details
- Grammarly and manual proofreading were used for final verification

A.5 Quantitative Contribution Estimate

Category	Human %	AI %
Architectural Analysis	100%	0%
Research & Reasoning	100%	0%
Drafting & Editing	~85%	~15%
Formatting & Clarity	~90%	~10%
Diagrams & Models	100%	0%

Overall estimated AI contribution: ≈ **10–15%**, consistent with A1 and A2.

A.6 Reflection

Across A1, A2, and A3, GPT-5 functioned as a reliable writing and formatting assistant. It improved clarity in longer sections, helped maintain consistent phrasing across multiple authors, and reduced time spent on stylistic revisions. However, the team found that architectural reasoning must remain entirely human-driven, as AI-generated explanations occasionally introduced subtle inaccuracies or oversimplifications.

When used with clear constraints and strict human oversight, the AI contributed positively to the workflow without compromising technical rigor or originality.