CS 2620 Final Project:
BLADE: Fast, Collaborative Markdown Editor with Math Typesetting

Michal Kurek*[1] and Natnael Teshome†[1]

[1]Harvard University

May 2025

1 Introduction

Courses at Harvard and other schools heavily involve group assignments. Students routinely co-author proofs and derivations in short bursts of synchronous editing. Popular platforms have critical shortcomings: plain old document editors often cannot render mathematics; Overleaf's heavyweight features that make writing math slow, with most of its features going unused. Our project asks a focused question:

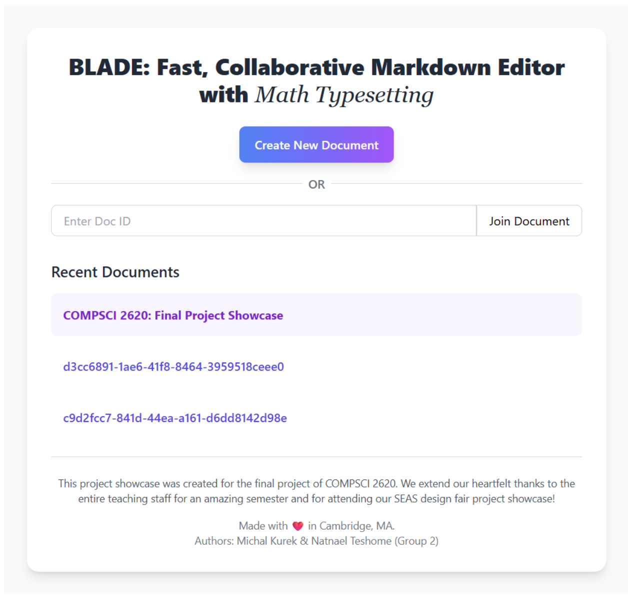
Can we offer a fast collaborative editor that is tailor-made for math typesetting?

This project, codenamed "Blade," was conceived to fill this gap. Our objective was to develop a collaborative editor specifically tailored for technical academic work, prioritizing speed, simplicity, and seamless integration of Markdown text and LaTeX-style math. The core deliverable is a web application deployable to a cloud platform (e.g., AWS EC2), allowing multiple users to join a shared document and experience real-time, concurrent editing. The central challenge in building such a system is ensuring consistency: how can we guarantee that all participants eventually see the exact same document state, even when multiple users edit the document simultaneously and network delays cause their actions to arrive at the server (and other clients) out of order?

This is a classic problem in distributed systems, often framed in terms of achieving eventual consistency. Concurrent edits can lead to conflicts analogous to merge conflicts in version control systems like Git, but these must be resolved automatically and in real-time without locking the document or interrupting the user experience.

Blade allows users to create a document and collaborate with multiple other users and share them with one another. Our proposed solution leverages Operational Transformation (OT), a family of algorithms specifically designed to manage concurrent edits in shared documents. Instead of relying on external libraries, a key pedagogical goal of this project for us was to implement a simple OT system from scratch to gain a deeper understanding of its intricacies. This report details the design, implementation, and evaluation of Blade, discussing the technical approach, the challenges encountered, and the key learning outcomes derived from this endeavor. For a live demo of our project, please refer to this Demo Link.

*mkurek@college.harvard.edu
†nteshome@college.harvard.edu



BLADE: Fast, Collaborative Markdown Editor with *Math Typesetting*

Recent Documents

COMPSCI 2620: Final Project Showcase

d3cc6891-1ae6-41f8-8464-3959518ceee0

c9d2fcc7-841d-44ea-a161-d6dd8142d98e

This project showcase was created for the final project of COMPSCI 2620. We extend our heartfelt thanks to the entire teaching staff for an amazing semester and for attending our SEAS design fair project showcase!

Made with ❤ in Cambridge, MA.
Authors: Michal Kurek & Natnael Teshome (Group 2)

Figure 1: The front page of Blade.

2 Operational Transform

Operational Transformation (OT) is a foundational technology for achieving consistency in optimistic concurrency control systems, particularly real-time collaborative editors where responsiveness is paramount. First introduced by C. Ellis and S. Gibbs in 1989, OT provides a framework for merging concurrent operations on a shared document state without resorting to traditional locking mechanisms (like locks around the document state), which would impede the *real-time* aspect of user experience.

In an OT-based system, every user action that modifies the document state—such as typing, deleting, or pasting—is encapsulated as an atomic *operation*. As described on our project poster, these operations typically represent either an insertion or a deletion and carry metadata crucial for processing: the target position (index) within the document where the change should occur, the content being inserted or the length of the text being deleted, and often version information (e.g., the document revision number the operation was based on). This granular representation allows user intentions to be mathematically manipulated.

The core challenge arises when operations generated concurrently by different users, based on the same document revision, arrive at different sites (or the central server) in potentially different orders due to network latency. If these operations were applied naively in the order they arrived, the document replicas would diverge. OT addresses this by defining rules to *transform* an incoming operation against any concurrent operations that have already been applied locally (or processed by the server). This transformation adjusts the incoming operation's parameters (like its target position or contents) to account for the effects of the concurrent changes, ensuring that the document state converges correctly across all replicas.

OT algorithms define transformation functions for pairs of operations, typically denoted abstractly as `transform(op1, op2)`. Given two operations, `op1` and `op2`, that were generated concurrently based on the same document state (revision `V`), this function produces a pair of new *transformed* operations, `[op1', op2']`. The key property these transformed operations must satisfy is *convergence*: applying the operations in different orders leads to the same final state (revision `V+2`). Formally:

$$\texttt{apply(apply(doc\_V, op1), op2')} = \texttt{apply(apply(doc\_V, op2), op1')}$$

even though

$$\texttt{apply(doc\_V, op1)} \neq \texttt{apply(doc\_V, op2)}$$

where `doc_V` is the document state at the revision `V` upon which both `op1` and `op2` were based, and function `apply(doc, op)` yields the new document state after applying operation `op` to `doc`. This convergence property ensures that, despite concurrency and network delays, all client replicas will eventually reach an identical document state once all operations have been processed.

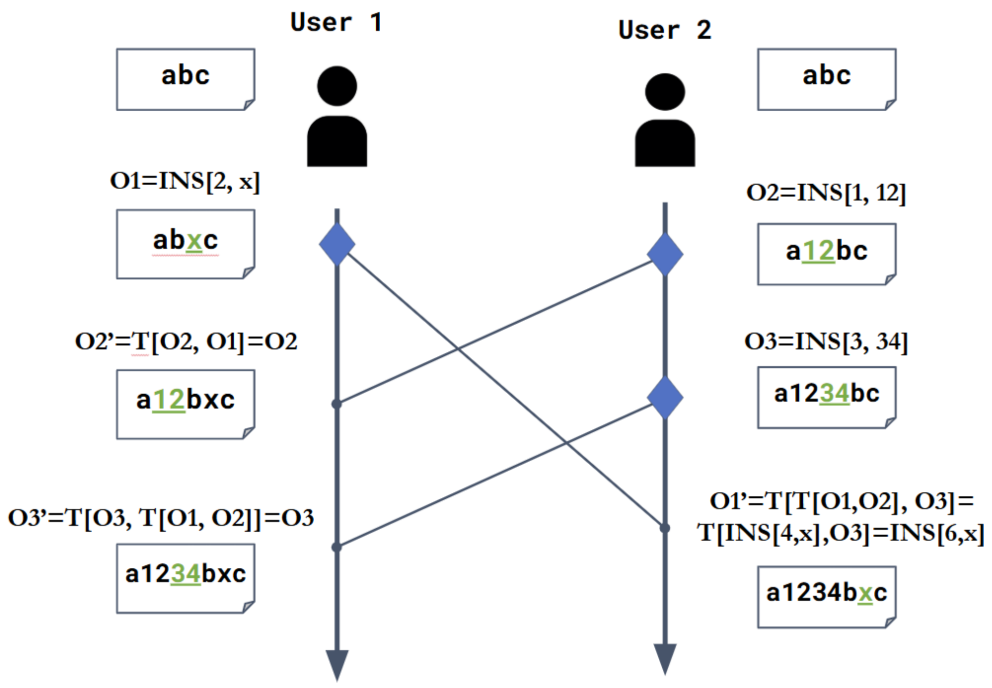


Figure 2: An example of OT.

Implementing a robust OT system that correctly handles all edge cases (e.g., overlapping deletions, adjacent insertions/deletions) is notoriously challenging. The transformation functions must satisfy strict mathematical properties beyond basic convergence, such as *intention preservation*—ensuring that the effect of an operation after transformation still reflects the user's original intent relative to the state they observed when creating the operation. As noted by a former Google Wave engineer involved in a complex OT system, "implementing OT sucks" (Source: `https://en.wikipedia.org/wiki/Operational_transformation`), highlighting the inherent, often underestimated, complexity involved in building correct and performant collaborative systems using this technique. Our project aimed to tackle this complexity head-on by implementing the core OT logic from scratch.

Figure 2 demonstrates how OT handles concurrent edits from two users to ensure eventual consistency. **Initial State:** Both User 1 and User 2 start with the document state `"abc"`. **Concurrent Operations:**

- User 1 generates `O1 = INS[2, x]`, intending to insert the string "x" at index 2 based on the initial document state. Locally, this results in User 1 seeing the document state `"abxc"`.
- User 2 generates `O2 = INS[1, 12]`, intending to insert "12" at index 1 based on the initial state. Locally, this results in `"a12bc"`.
- User 2 then generates another operation, `O3 = INS[3, 34]`, based on their local state *after* applying O2 (`"a12bc"`). This results locally in `"a1234bc"`.

Transformation and Integration (User 1's Replica):

1. User 1 receives `O2`. It must be transformed against `O1`, which was applied locally. The transformation `T[O2, O1]` yields `O2'`. In this case, `O2'` happens to be identical to `O2` (`INS[1, 12]`) because O1's insertion point (index 2) does not affect O2's insertion point (index 1). Applying `O2'` to User 1's state `"abxc"` results in `"a12bxc"`.

2. User 1 then receives `O3`. `O3` must be transformed against the operations already processed on User 1's replica in their logical order (effectively, against the combined effect of O1 and O2'). The transformation is `O3' = T[O3, T[O1, O2]]`, which results in `O3'` being identical to `O3` (`INS[3, 34]`). Applying `O3'` to User 1's state `"a12bxc"` results in the final state `"a1234bxc"`.

Transformation and Integration (User 2's Replica):

1. User 2 has already applied `O2` and `O3` locally, reaching state `"a1234bc"`.
2. User 2 receives `O1`. `O1` has been generated on an old revision of the document state (from User 2's perspective) and hence must be transformed against the operations User 2 has already applied (`O2` followed by `O3`). This requires transforming `O1` against `O2` first (`T[O1, O2]` results in `INS[4,x]`, shifting O1 past O2's insertion), and then transforming that result against `O3`. The transformation `T[INS[4,x], O3]` results in `O1' = INS[6,x]` (shifting the operation past O3's insertion "34").
3. Applying the fully transformed `O1'` to User 2's state `"a1234bc"` results in the final state `"a1234bxc"`.

Convergence: Despite the different sequence of local edits and received operations, both users' replicas converge to the identical final document state `"a1234bxc"`, demonstrating the core purpose of OT in maintaining consistency in collaborative editing.

3 System Design (How It Works):

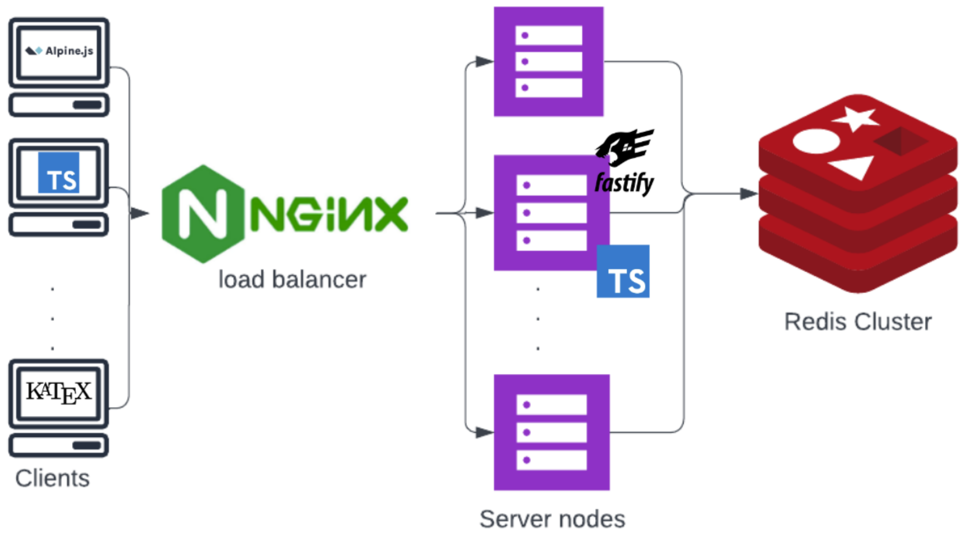
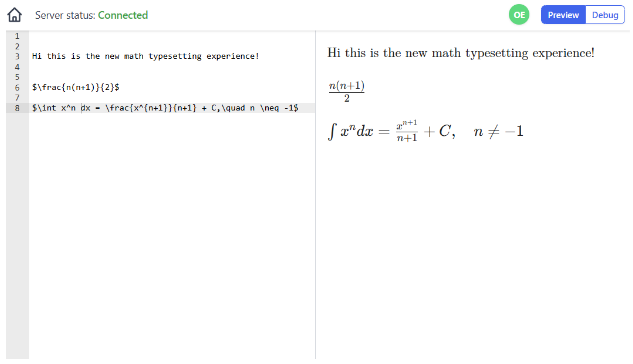


Figure 3: High level system design diagram.

Blade employs a client-server architecture optimized for real-time collaborative editing. The design choices and implementation details were guided by the core goals of creating a *fast*, *lightweight* editor with robust math support, while fulfilling the pedagogical objective of implementing the Operational Transformation (OT) logic from scratch.

3.1 Overall Architecture

Web browser clients connect via persistent WebSocket connections, managed by the Socket.IO library, to a scalable backend infrastructure. An Nginx instance acts as a reverse proxy and programmable load balancer, routing connections to one of multiple backend Node.js server nodes. These nodes handle application logic and OT processing, interacting with a Redis Cluster instance for persistence. The use of TypeScript across the stack allows sharing critical code, most notably the OT library (`ot.ts`), between the frontend and backend codebases.



```


Hi this is the new math typesetting experience!


$\frac{n(n+1)}{2}$

$\int x^n dx = \frac{x^{n+1}}{n+1} + C,\quad n \neq -1$
```

Hi this is the new math typesetting experience!

$\frac{n(n+1)}{2}$

$\int x^n dx = \frac{x^{n+1}}{n+1} + C,\quad n \neq -1$

Figure 4: A document in Blade.

3.2 Frontend (Client)

The client-side is a Single Page Application (SPA), primarily implemented within `client.ts`, designed for responsiveness and a seamless user experience:

- **Core Logic (TypeScript):** TypeScript forms the backbone of `client.ts`, chosen for its type-safety and our familiarity with it. The code implements the client's state machine logic (managing `state`, `outstandingOp`, `bufferedOp` variables to track synchronization status, see our GitHub codebase for reference), handles events received from the web editor exposed to the user and the Socket.IO connection, and orchestrates the overall application flow, including when to push changes or request history from the backend server.

- **UI Framework (Alpine.js):** Integrated directly into the HTML (e.g., `index.html`, `dashboard.html`), Alpine.js provides lightweight UI reactivity. It is not as heavyweight as the obvious alternatives (React, Svelte, Vue.js) and was a pleasure to work with. It declaratively binds HTML elements to the state defined within the `editorApp` function in `client.ts`. Examples include displaying connection status via `statusText`, rendering the user list from the `users` array using `x-for`, and managing UI views. Its very minimalistic and was easy to integrate with our core logic mentioned above.

- **Editor Core (Ace Editor):** The primary user interface for text entry, instantiated via `ace.edit("editor")` in `client.ts`. Ace is a web component that exposes a rich API, which we heavily utilized: it exposes methods such as e.g., `editor.setValue()`, which programmatically updates the editor content when receiving server updates (within the `withNoLocalUpdate` helper to prevent feedback loops); the `editor.session.on("change", delta => ...)` listener captures user input as `delta` objects, which are processed by `handleLocalDelta` and converted to `TextOperations` using the `convertAceDeltaToOp` helper; cursor position is managed using `editor.getCursorPosition()` and `moveCursorToPosition()`, interacting with the OT library's `transformCursorIndex` method to preserve user cursor location across remote edits, solving a key usability issue. Ace was chosen for its maturity and necessary programmatic control.

- **Rendering (Markdown + KaTeX):** User input in Ace is interpreted as Markdown with embedded LaTeX math. A Markdown parser converts the Markdown text to HTML for the preview pane. Within this process, a math rendering library called KaTeX renders any LaTeX math expressions (e.g., between `$` or `$$`) into high-quality mathematical notation. KaTeX was chosen primarily for its rendering speed, essential for a real-time preview, and excellent typesetting quality.

- **Communication (`socket.io-client`):** This library manages the WebSocket connection, established in `client.ts` via `io(window.location.origin, { query: { docId } })`, passing the document

ID for routing. It simplifies handling connection lifecycle events (`on('connect')`, `on('disconnect')`) and processing messages from the server through various listeners like `socket.on('initial_state', ...)`, `socket.on('ack', ...)`, `socket.on('update', ...)`, `socket.on('user_joined', ...)`, etc. Client actions are sent via `socket.emit('push', pushMsg)` and `socket.emit('pull', pullMsg)`, sending serialized operations or revision numbers as defined in `types.ts`.

- **Local OT Logic (`client.ts` using `ot.ts`):** To optimize performance and bandwidth, local edits captured by `handleLocalDelta` are converted to `TextOperations` and composed with any pending local changes stored in `bufferedOp` using `this.bufferedOp.compose(op)`. This batches edits. When pushing, `bufferedOp` is moved to `outstandingOp`. Crucially, when receiving server updates (`on('update')`), the incoming operation is transformed against both `outstandingOp` and `bufferedOp` using `TextOperation.transform` before being applied locally, ensuring correctness despite latency and concurrency.

3.3 Backend (Server Nodes)

The backend, primarily implemented in `server.ts`, consists of potentially multiple identical server nodes using Node.js and specialized libraries:

- **Runtime & Framework (Bun.js + Fastify):** Node.js provides the asynchronous, event-driven runtime (with the Bun JS runtime offering some performance benefits over the regular Node.js runtime), suitable for handling many concurrent WebSockets. Fastify (`fastify()`) is used as a high-performance web framework to structure the application. Its role in `server.ts` includes handling static frontend/backend file serving, CORS permissions, and crucially, the integration with the client-side Socket.IO library. It also defines the necessary HTTP routes (`app.get('/')` for dashboard, `app.get('/docs/:docId')` for the core editor shell, `app.post('/docs')` for creating new documents) and starts the HTTP/WebSocket server via `app.listen()`.

- **Responsibilities:** Each node handles WebSocket connections routed by Nginx. Within the main `app.io.on('connection', (socket) => ...)` handler, it manages the state for its assigned documents, processes incoming client operations (`push` messages) using the shared `ot.ts` library, applies validated and transformed operations, broadcasts updates (`update` messages) to other clients in the same document room, and persists state changes to Redis.

- **WebSocket Handling (`socket.io` Server-Side):** Integrated via `fastifyIO`, Socket.IO manages connections and rooms (like namespaces, but for websockets). Key actions within the connection handler include: joining a socket to a document-specific room (`socket.join(docId)`), listening for client messages (`socket.on('push', async (msg) => ...)`, `socket.on('pull', ...)`), emitting direct responses (`socket.emit('ack', ackMsg)`), and broadcasting new server updates to others in the room (`socket.to(docId).emit('update', updateMsg)` or `socket.broadcast.emit('user_joined', ...)`). This room feature is fundamental to efficiently targeting broadcasts in the multi-document architecture to only a specific subset of clients connected to a particular set of documents.

- **Multi-Document State:** Each server node maintains an in-memory hashmap (`Map<string, DocumentData>`) where `DocumentData` is the minimal necessary state required to track the document contents as well as any changes made to it in recent revisions: { `content: string; revision: number; history: TextOperation[]` }. This allows a single Node process to concurrently manage the state, revision history, and OT context for multiple distinct documents routed to it. The `history` array is essential for the OT `transform` function when processing new client operations.

- **Persistence (Redis via `bun:redis`):** Redis (accessed via the `bun:redis` client in `server.ts`) stores the authoritative, persisted state for each document, keyed by `doc:<docId>`. On startup, the server attempts to load existing documents using `redis.keys("doc:*")` and `redis.get(key)`, deserializing the JSON string containing `content`, `revision`, and `history`. After processing a `push` message and applying the transformed operation, the updated state (including the new operation added to `history`) is asynchronously saved back using `redis.set(doc:${docId}, JSON.stringify(...))`. Redis was chosen for its speed and trivial ease of integration into our application, with Redis Cluster providing a path to persistence scalability.

3.4 Operational Transformation (OT) Library (`ot.ts`)

This is the intellectual core, implemented from scratch in TypeScript. It defines the `TextOperation` class encapsulating retain, insert, and delete components. Key methods are:

- `apply(doc)`: Applies an operation to a string (used by server to update `content`, used by client to update `virtualDoc`).
- `compose(other)`: Composes sequential operations (used by client in `handleLocalDelta` to buffer edits).
- `transform(op1, op2)`: The static method implementing the core OT logic to resolve concurrent operations (used heavily by server in the `on('push')` handler to transform incoming ops against history, and by client in `on('update')` and `on('history')` handlers to transform server ops against local buffers).
- `transformCursorIndex(index)`: Calculates resulting cursor position after an operation (used by client in `on('update')` to preserve cursor).
- `fromJSON(ops)`, `toJSON()`: For serialization over the network (used when sending/receiving messages defined in `types.ts`).

Being written in TypeScript ensures type-safety on both client and server. Please refer to the implementation and, more importantly, the corresponding tests for example usage of our developed library.

3.5 Distributed Strategy and Load Balancing

A sharded strategy simplifies OT consistency:

- **Single Source of Truth per Document:** Responsibility for a `docId` is assigned to exactly one backend Node.js process, which serializes operations for that document.
- **Nginx Hash-Based Routing:** We extend our deployed Nginx config with a Lua script (example in `nginx.conf` to hash the `docId` from the WebSocket request URI and route the connection deterministically (`hash % num_nodes`) to the assigned backend node. This partitions documents and ensures all clients for a document talk to the same server process.
- **Redis Cluster for Persistence:** Backend nodes persist state of each document (`content`, `revision`, `history`) to Redis Cluster under keys like `doc:<docId>`, as implemented in `server.ts` using `redis.set`. This occurs *after* successful processing. Redis Cluster shards keys, distributing persistence load. Recovery involves reloading this state from Redis on node restart.
- **Benefits:** This strategy simplifies OT logic drastically, distributes load across backend and Redis nodes, and provides clear state management per node.

3.6 Communication Protocol (WebSockets)

The underlying protocol enabling real-time, persistent, bidirectional communication between client (`client.ts`) and server (`server.ts`), managed via the Socket.IO library. Its low latency and server-push capability are essential for collaboration, superior to HTTP polling. The specific message formats exchanged are strictly defined by TypeScript interfaces in `types.ts` (e.g., `ClientPushMsg`, `ServerAckMsg`, `ServerUpdateMsg`, `ServerYourIdentityMsg`). This was inspired by our previous homeworks/demos in COMPSCI 2620.

4 What We Learned

This project gave us substantial learning across theoretical concepts, practical implementation techniques, and software engineering practices. We gained very important insights about distributed systems.

1. **Deep Dive into Operational Transformation (OT):** While writing the proposal, we expected OT to be a complicated algorithm, the implementation of the algorithm to fit our specific use case proved to be even more complicated than what we anticipated:
   - The intricate logic required for the `transform(op1, op2)` function to handle various combinations of concurrent inserts, deletes, and retains, ensuring the convergence property:

     `apply(apply(doc, op1), op2’)` == `apply(apply(doc, op2), op1’)`
   - The importance of careful state representation (`baseLength`, `targetLength`) within operations for validation and correct transformation.
   - The non-trivial nature of extending OT, exemplified by the work needed to implement cursor position transformation (`transformCursorIndex`) to fix a bug we encountered where the cursor position of different users would be reset by operations of a single user.
   - Why OT implementations are considered notoriously difficult (”implementing OT sucks”) – seemingly small edge cases can easily lead to divergence if not handled meticulously.
2. **Mastering Client-Side State Management for Real-Time Sync:** The core challenge of maintaining consistency despite concurrency and latency manifested primarily in the client-side logic (Code: `client.ts`). We learned:
   - The necessity of a robust client-side state machine (`state`: `synchronized`, vs `dirty`, `awaiting ACK`, etc.) to correctly handle the asynchronous arrival of server messages (`ack`, `update`) relative to local user actions.
   - How to correctly manage buffered local operations (`bufferedOp`) and operations awaiting server acknowledgment (`outstandingOp`), including composing local changes and transforming incoming server updates against both buffers.
   - The practical implementation of optimistic concurrency: allowing immediate local edits while correctly merging server updates in the background requires careful transformation logic on the client.
3. **Practical Distributed System Design Choices:** The proposal outlined a sharded approach, and when we gained a lot of knowledge about sharding in practice:
   - The value of simplifying the consistency problem by assigning a single source of truth per document to a specific backend node, avoiding the complexities of fully distributed OT.
   - How to leverage infrastructure tools like Nginx with Lua scripting for stateful routing (hash-based routing by `docId`) to enforce this sharding strategy.
   - Integrating backend state management (`Map<string, DocumentData>` in `server.ts`) with persistence (Redis) and client communication (Socket.IO rooms per document ID) to support multiple concurrent documents.
4. **Real-Time Communication Protocols and UX:** Building a system intended to feel ”instant” involved learning:
   - Effective use of WebSockets via Socket.IO for low-latency, bidirectional communication, including designing a clear message protocol (`types.ts`) for actions like `push`, `pull`, `ack`, `update`.
   - The importance of features beyond core editing for collaborative UX, such as visualizing user presence (anonymous names, join/leave notifications).

- Balancing responsiveness and network efficiency through client-side operation buffering (`bufferedOp`) and considering sync frequency.

5. **Full-Stack Integration Challenges:** Connecting all the pieces required understanding interactions across the entire stack:
   - Coordinating frontend state (Alpine.js UI, Ace editor content) with backend state (Node.js/Fastify server, Redis).
   - Managing dependencies like rendering Markdown and math (KaTeX).
   - Ensuring consistent data representation and types across client and server, showing the value of TypeScript and shared interfaces (`types.ts`).

6. **Importance of Testing and Debugging in Concurrent Systems:** The project starkly showed the limitations of manual testing for complex, concurrent systems:
   - Bugs related to race conditions or specific timing of asynchronous events are hard to reproduce manually.
   - The recognized need for more rigorous testing methodologies, including automated testing, potentially fuzz testing or invariant checking for the OT library, and comprehensive integration tests.
   - The value of clear logging and potentially runtime assertions for diagnosing issues in distributed environments.
   - The added value of a type safe language (we used TypeScript as opposed to JavaScript).
   - The need for clear documentation to track system state and endpoints.

7. **Managing Tradeoffs**:
   - We learned that making tradeoffs about what is possible to implement in a short time vs long time frame is essential to building a large scale project
   - OT has very rich features and implementing some of the seemingly straightforward additions like redo and undo can be extremely tricky and may not fit the timeline of this project even if we work extremely hard. Hence, we learned that before implementing a new feature, we have to plan in advance how much time and effort it takes and evaluate the tradeoff between how important the feature is and how valuable our time is and thus spend more time on other more important and relatively easier to implement features.
   - In this way, we found this assignment to be very different from the design exercises we did throughout the course. Because in the design exercises, we know that the design requirements outlined by the course are meant to fit into the time available for us to complete. But, in the final project, since we all are doing our own specific projects, we are responsible for figuring out what is plausible and what is not.
   - We believe this is a very important skill any software engineer should possess.

8. **Testing**: We wrote rigorous tests for our core business logic.
   - Ensured our OT works very smoothly not only using tests but also running rigorous live demos.
   - We perform a decent amount of randomized/fuzzy testing of the OT library we developed.

9. **Deploying**:
   - Setting up a virtual private server (VPS) from scratch. Setting up a domain name, A records, firewall, remote deployment through ssh, setting up a reverse proxy (Nginx), creating new `systemd` jobs on Linux.
   - In general, we learned a lot not only about how to *build* a distributed system, but also how to *deploy* one too.

10. **Presenting**:

    - We learned how to make an effective poster and showcase it to people and answer live questions.
    - We did a live demo of our working application during the SEAS Design Fair.

11. **Working in a team:**:

    - Throughout this final project, our team members collaborated significantly from the ideation of the project, to the design, implementation, poster making, and report write-up. We believe developing these skills is an essential part of the software engineering that we will encounter in the real world.

In summary, we met all our proposal goals and made our MVP more robust than we had ever imagined. In addition, we implemented some cute feature user aliases, and mobile-responsive front-end among other things. Check out our demo to experience the prototype of Blade first-hand.

Thank you for your support throughout the course!