eSim Summer Fellowship 2026

On

Real-Time Live Collaboration and Google OAuth Authentication Hardening in eSim on Cloud

Submitted by

Riya Mehta

Department of Computer Science and Engineering

VIT Bhopal University

Under the guidance of

Prof. Prabhu Ramachandran

Principal Investigator

Department of Aerospace Engineering

Indian Institute of Technology Bombay

July 25, 2026

Acknowledgment

I express my sincere gratitude to Prof. Prabhu Ramachandran for providing me with the opportunity to be part of the FOSSEE internship program and for his continued efforts in promoting open-source engineering tool development. His leadership and vision have been instrumental in fostering meaningful student participation in the open-source ecosystem.

I also acknowledge Prof. Kannan M. Moudgalya for his foundational role in establishing and strengthening the FOSSEE initiative. His contributions to open-source education and the development of the FOSSEE fellowship framework have been pivotal in creating the academic and organisational platform through which this internship was undertaken.

My sincere appreciation extends to my mentor, Mr. Sumanto Kar, for his continual support, technical guidance, and encouragement throughout the duration of this project. His insights and feedback played a key role in refining ideas, overcoming challenges, and ensuring timely completion of the tasks assigned to me.

I would also like to thank my internal mentors, Mr. Varad Patil and Ms. Shanthi Priya K, for their valuable guidance, coordination, and technical inputs during the internship. Their mentorship contributed significantly to the clarity, progress, and successful execution of the work.

I would also like to thank my fellow FOSSEE interns and teammates on the eSim on Cloud platform for their close collaboration during this internship. The two subsystems described in this report the real-time Live Collaboration layer in the schematic editor and the hardening of the Google OAuth authentication flow both sit at the boundary between the existing Django backend, the React front end, and the schematic-editor internals, and their correct behaviour depended on repeated, patient interface discussions with the contributors who maintain those surrounding components. I am grateful for their cooperation as the shared code evolved.

This internship has been an enriching learning experience, allowing me to work closely with open-source EDA tools, real-time web technologies, and authentication systems, and to gain exposure to real-world collaborative software workflows. The

knowledge acquired during this period will undoubtedly support my future academic and professional pursuits.

I would also like to thank the entire FOSSEE team for their coordination, assistance, and timely interactions at various stages of this work. Their collective efforts ensured smooth workflow, resource accessibility, and effective project execution.

The two components on which I contributed most directly during this fellowship, and which form the technical core of this report, are: (i) the Live Collaboration subsystem for the eSim on Cloud schematic editor, comprising real-time presence and live shared viewing over WebSockets; and (ii) the reliability and security hardening of the Google OAuth 2.0 sign-in flow, including a corrected token-exchange path, structured error handling, and a dedicated automated test suite.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

Cloud-hosted engineering tools are increasingly expected to behave less like single-user desktop applications that happen to run in a browser and more like genuinely shared, multi-user environments. Two capabilities are central to that expectation. The first is real-time collaboration: the ability for more than one person to look at, and eventually work on, the same artefact at the same time, seeing each other's presence and each other's changes without manually reloading or re-sharing files. The second is dependable authentication: a sign-in path that is not only convenient ideally a single click through an identity a learner already owns, such as a Google account but also robust, so that a misconfigured credential, a cancelled consent screen, or a dropped network request produces a clear, recoverable outcome rather than a blank page or a silently broken session.

Circuit-design education is a representative setting for both. An electronic design automation (EDA) classroom, a laboratory demonstration, or a peer-learning session frequently involves an instructor wanting to walk a group of learners through a schematic live, or two learners wanting to inspect the same circuit together while discussing it. Until such a capability exists, the only options are screen-sharing over a separate video tool, or each learner independently opening a copy of the design and losing any sense of a shared, synchronised view. At the same time, every one of those learners must first get into the platform, and for a browser-based tool aimed at large classroom and MOOC audiences, the smoothness and reliability of that very first sign-in step disproportionately shapes whether a learner ever reaches the circuit at all.

This report documents work carried out during the FOSSEE eSim Summer Fellowship on exactly these two capabilities within eSim on Cloud, the browser-hosted variant of the open-source EDA tool eSim. The contribution has two distinct but complementary parts. The first is a Live Collaboration subsystem for the schematic editor, delivered in two phases: a presence layer that lets users opening the same shared project see one another in real time, and a live shared viewing layer that streams the editor's schematic to every other participant so that viewers watch the owner's canvas update as it happens. The second is a hardening of the Google OAuth 2.0 authentication flow, which repaired a set of latent defects in the sign-in path, added structured error handling and logging at every failure point, corrected the token-issuing strategy, and introduced a dedicated automated test suite so that the sign-in behaviour is now verifiable in isolation rather than only by manual trial.

The two parts are presented together because they were the author's two major areas of contribution during the fellowship, and because they share a common engineering theme that recurs throughout this report: both are concerned with what happens at the boundaries between independently developed components between the browser and the server, between the React front end and the Django backend, between an external identity provider and the local user model and both required designing for the cases where those boundaries behave imperfectly, such as a WebSocket that drops mid-session or an OAuth callback that arrives with missing parameters.

It is worth being precise about terminology at the outset. In the collaboration subsystem, a presence session denotes the period during which a user has a schematic open and is therefore counted as online in that project's room; this is tracked over one WebSocket connection. A viewing session denotes the parallel period during which that same user is receiving live schematic snapshots from whoever currently owns the project; this is tracked over a second, deliberately separate WebSocket connection. Keeping these two notions and their two connections isolated from one another is a central design decision described in Chapter 4. In the authentication work, by contrast, a login session means the conventional web notion: the authenticated identity a user holds after a successful sign-in, represented by a token stored in the browser. The three terms are related but distinct, and this report is careful to keep them separate.

1.2 Overview of eSim

The Free and Open Source Software in Science and Engineering Education (FOSSEE) project, hosted at IIT Bombay, promotes tools that students and professionals can inspect, extend, and redistribute. Among these, eSim is an open-source electronic design automation environment oriented toward circuit capture, simulation using NGSPICE, and related learning workflows. It occupies an important place in the Indian academic ecosystem because it lowers the barrier to structured experimentation with circuit behaviour without dependence on proprietary licences. Students who train on eSim learn to think in terms of schematics, netlists, device models, and simulation discipline, which transfers well to other tools in the same conceptual family.

eSim on Cloud extends this mission by removing the requirement to install eSim and its simulation dependencies on a local machine. A learner can open a browser, draw or upload a circuit in the schematic editor, and receive simulation results without configuring a desktop toolchain. The schematic editor itself is built on a JavaScript graph library (mxGraph), wrapped in a React application, with a Django backend that persists saved projects, exposes the simulation and library APIs, and handles user accounts and authentication. Saved projects are addressable by a save identifier together with a version and branch, and can be shared by URL a fact that the Live Collaboration subsystem builds directly upon, since a shared project URL is precisely what two users must both open to find themselves in the same collaboration room.

Before this fellowship's contribution, eSim on Cloud presented each learner with an essentially solitary editing experience: even when two people opened the same shared project URL, neither was aware of the other, and neither saw the other's changes without a manual save-and-reload cycle. Separately, the platform's Google sign-in path, while functional in the common case, contained several latent defects that surfaced only when something went wrong a missing credential, a partial callback, or a previously unverified account at which point the failure was opaque to the learner. The work described in this report addresses both of these gaps.

The connection to the wider eSim project is direct in both cases. The Live Collaboration subsystem operates on exactly the same schematic model (mxGraphModel XML) that eSim on Cloud already saves and simulates, so a collaboratively viewed circuit is the same artefact a learner would otherwise edit and simulate alone. The authentication

work governs the front door through which every learner whether they go on to simulate, collaborate, or simply browse the component library must pass. Neither subsystem changes what a circuit is or how it is simulated; both change how many people can engage with it, and how reliably they can get to it.

1.3 Objectives of the Project

The central objective of the fellowship was to make eSim on Cloud measurably more usable as a shared, multi-user platform along two axes collaboration and authentication without disturbing the existing single-user editing, saving, and simulation workflows that learners already depended on. This central objective was decomposed into a set of concrete, individually testable sub-objectives across the two contribution areas.

Live Collaboration objectives

1. **Presence rooms.** Establish a per-project presence room that is created automatically when a saved project is opened, so that every user viewing the same shared project is grouped together without any explicit join step.
2. **Real-time online awareness.** Broadcast join and leave events over a dedicated presence WebSocket so that each participant sees an accurate, live list of who else is currently present, together with an online count and per-user connection indicator.
3. **Correct multi-tab and anonymous handling.** Ensure an authenticated user opening the same project in several browser tabs is counted exactly once, while an anonymous guest is tracked per connection, so that the online list reflects distinct people rather than distinct tabs.
4. **Live shared viewing.** Establish a second, separate synchronisation WebSocket that streams the owner's full schematic to all other participants, so that viewers watch the schematic update in real time as the owner edits.
5. **Editor / viewer role separation.** Compute each participant's role from the project owner and the authenticated identity, grant the owner a fully editable canvas, and place everyone else in a constrained viewer mode in which selection, movement, deletion, connection, and sidebar drag-and-drop are all blocked while panning and zooming remain available.

6. **Non-destructive snapshot application.** Apply each incoming snapshot on a viewer's canvas without resetting that viewer's own zoom level or pan position, so that a viewer can explore the schematic independently while still receiving live content updates.
7. **Resilience.** Debounce the editor's outgoing snapshots so that a burst of edits produces one update rather than many, exclude the sender from its own broadcast so the editor never re-renders its own canvas, and reconnect either WebSocket automatically with exponential backoff if the connection drops.
8. **Non-invasive integration.** Add all of the above beneath the existing schematic editor, saving, and simulation code paths, so that a learner working alone sees no change whatsoever in existing behaviour, and a deep-linkable share URL correctly carries the version and branch needed to reload the exact schematic.

Authentication-hardening objectives

9. **Correct the callback control flow.** Repair the Google OAuth callback so that a token exchange is attempted only when both the state and code parameters are present, rather than when either one happens to be present, eliminating a class of unhandled exceptions on partial callbacks.
10. **Fail safely on misconfiguration.** Detect absent or empty OAuth credentials before any network call is made, and return a clear, user-facing message rather than allowing the request to proceed and crash deep inside the token-exchange library.
11. **Structured error handling end to end.** Wrap each externally dependent step token exchange, user-info retrieval, and user/token persistence in its own guarded block, so that a failure at any single step redirects the user cleanly back to login with an informative message and leaves no half-created user behind.
12. **Repair the token strategy.** Replace the broken token-issuing code, which depended on a module removed in modern Django and created a token against an unfilled placeholder user, with a correct get-or-create against the real authenticated user.
13. **Activate verified-by-Google accounts.** Treat a successful Google sign-in as sufficient proof of identity to activate an account that had been created via the sign-up form but never had its email verified, resolving a previously dead-ended state.

14. **Clean client-side recovery.** On the browser side, clear any stale token on a failed callback and surface a specific Google sign-in error message, so a failed attempt does not leave the app holding an invalid token.
15. **Regression protection.** Introduce a dedicated automated test suite covering parameter validation, missing credentials, token-exchange failure, the successful new-user and returning-user paths, inactive-account activation, and the inactive-login error message, so that all of the above behaviours are verifiable without manual sign-in attempts.

Taken together, these sub-objectives compose into two coherent guarantees rather than a list of unrelated features. For collaboration: at every moment, every participant in a shared project should see an accurate picture of who else is present and, if they are not the owner, a faithful and current view of the owner's schematic, and this picture should converge back to correctness within a bounded time even after a dropped connection. For authentication: every sign-in attempt should terminate in exactly one of two clearly distinguishable states an authenticated session, or an informative error returned to the login page with no unhandled exception, no orphaned half-created user, and no stale token left behind in between.

1.4 Methodology

The methodology followed an iterative, boundary-first development cycle. For each subsystem, development began by identifying the exact interface at which the new code would meet existing code the WebSocket URL and message shapes for collaboration, the OAuth callback view signature and its rendered template for authentication and then building outward from that interface, validating each increment against a concrete, reproducible scenario before moving on. The emphasis throughout was on making failure modes visible: a dropped socket, a missing OAuth parameter, or a viewer receiving a snapshot mid-edit should each produce an observable, predictable result rather than a silent inconsistency.

1.4.1 Presence Layer First

Development of the collaboration subsystem began with the presence layer in isolation, because presence is the simpler of the two real-time concerns and because it establishes the room abstraction that live viewing later reuses. A minimal

PresenceConsumer was built that accepted a WebSocket connection on /ws/presence/<save_id>/, added the connection to a per-project channel group, and broadcast a join event. This was tested with two browser tabs pointed at the same shared URL before any user-list rendering existed, using the browser's developer tools to confirm that the join frame arrived on the second tab. Only once the raw broadcast was confirmed working was the Redux-backed PresencePanel UI, with its online list and join/leave toasts, layered on top.

1.4.2 Redis-Backed Presence State

Because a channel group alone does not answer the question who, as a distinct person, is currently in this room, a small Redis-backed PresenceStore was introduced alongside the channel layer. It maintains two hashes per project, one mapping each live connection (channel name) to a user key, and one mapping each user key to that user's displayed information, both carrying a twenty-four-hour time-to-live as a safety expiry. This store is what makes multi-tab de-duplication and clean last-tab-leave detection possible, and it was deliberately kept separate from Django Channels' own group management so that the two never had to be reasoned about together.

1.4.3 Live Viewing as a Second, Isolated Channel

The live viewing layer was then built as an entirely separate WebSocket on /ws/sync/<save_id>/, handled by a distinct SyncConsumer on a distinct channel group. This isolation was a conscious decision made early: presence and synchronisation have different lifetimes, different failure tolerances, and different message rates, and coupling them would have made each harder to reason about. The synchronisation consumer's connect handler deliberately has no presence side-effects, and the two groups never cross-post.

1.4.4 Snapshot Debouncing and Sender Exclusion

An early prototype broadcast a snapshot on every single graph-change event, which produced a storm of redundant messages during a drag and, worse, caused the editor to receive and re-render its own changes. Two refinements resolved this: a 300 ms debounce so that a continuous burst of edits collapses into a single snapshot once the editor pauses, and a sender-exclusion rule in the relay so that the originating editor is skipped when the snapshot is fanned out. Viewers apply the snapshot through the editor's existing gallery-render routine, which was found to preserve the viewer's own zoom and pan because it

clears and redraws only the child cells, never touching the view's scale or translation.

1.4.5 Authentication: Reproduce, Then Repair

The authentication work followed a strict reproduce-then-repair discipline. Each defect was first captured as a failing automated test that exercised the callback view directly with a crafted request missing state, missing code, empty credentials, a mocked token-exchange exception, a user-info response with no email, and so on so that the exact failing behaviour was pinned down before any fix was written. Only then was the view rewritten, one guarded step at a time, until every previously failing test passed. This ordering mattered because several of the defects were latent: they never fired on a happy-path sign-in and would otherwise have been easy to fix without evidence that the fix addressed the real fault.

1.4.6 Testing Strategy

Validation relied on a combination of scenario-based manual testing and automated tests. For collaboration, a two-participant flow (owner plus viewer) and a multi-participant flow were exercised across separate browsers and tabs, with the browser developer tools' WebSocket inspector used to confirm the exact frames crossing each connection, and a deliberate mid-session network drop used to confirm reconnection. Each check in the Phase 1 and Phase 2 validation matrices of Chapter 4 corresponds to one such reproducible scenario. For authentication, the dedicated test suite described above was run on every change, and its scenarios together with their exact inputs are tabulated in Chapter 4. Figure 1.1 summarises how the two contributions sit within the overall eSim on Cloud request flow.

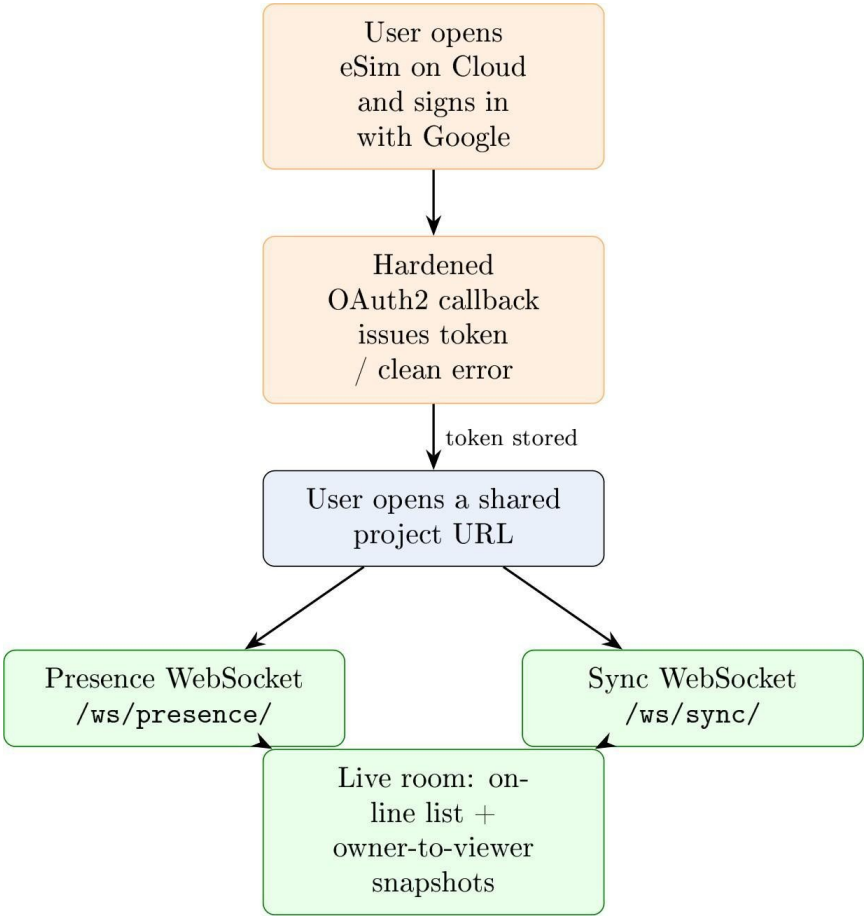

Figure 1.1: Overall placement of the two contributions within eSim on Cloud. The hardened Google OAuth flow (orange) governs sign-in; once inside, opening a shared project URL establishes the two independent Live Collaboration WebSockets (green) that together form a live room.

Figure 1.1 makes the division of responsibility explicit: the authentication work owns the sign-in boundary, while the collaboration work owns everything that happens once two authenticated (or anonymous) users find themselves in the same project. The remainder of this report treats each contribution in turn, after first surveying the relevant literature and stating the problem precisely.

Chapter 2

Literature Survey

The two contributions in this report draw on two established bodies of practice: real-time collaborative web systems on the one hand, and secure delegated authentication on the other. This chapter reviews the concepts and design guidance that shaped the implementation, organised by theme, and notes at each point how the guidance was applied.

2.1 Real-Time Collaboration over WebSockets

Real-time collaborative applications require a persistent, bidirectional channel between browser and server, because the defining characteristic of collaboration is that the server must be able to push another user's activity to a client without that client having asked for it. The WebSocket protocol is the standard mechanism for this, providing a single long-lived, full-duplex connection over which small framed messages flow in both directions. The literature consistently contrasts this with polling approaches, noting that polling either wastes requests when nothing has changed or introduces latency when the poll interval is long, whereas a WebSocket delivers an event essentially as soon as it occurs. Both real-time layers in this project presence and live viewing are built on WebSockets for precisely this reason.

A recurring theme in WebSocket literature is that a long-lived connection needs an application-level liveness signal, because the transport's own keep-alive confirms only that the socket is open, not that the application behind it is healthy and responsive. The recommended pattern is a small periodic ping/pong exchange at the application layer. This project follows that recommendation on both channels: each client sends a lightweight PING on a fixed interval and the server replies PONG, so that a half-open connection is detected and can trigger the reconnection logic described below.

2.2 Presence Systems and Room Abstractions

Presence the awareness of who else is currently here is a well-studied building block of collaborative systems, from chat applications to shared document editors. The literature describes presence in terms of rooms (logical groupings of connected clients) and membership events (join and leave notifications broadcast to a room). Server-side frameworks for real-time web applications, including Django Channels used here, provide a channel-group abstraction that maps naturally onto this room model: adding a connection to a group and broadcasting to that group are primitive operations.

However, the literature is also clear that a channel group answers which connections are in this room but not which distinct people are in this room, and that the gap between the two matters whenever a single person may hold several connections at once most commonly by opening the same page in multiple browser tabs. The standard remedy is to maintain an explicit membership store keyed by a stable per-user identity rather than by connection, and to derive the visible participant list from that store. This project adopts exactly this pattern: a Redis-backed presence store keyed by a user key de-duplicates multiple tabs of the same authenticated user into a single visible participant, while still tracking anonymous guests per connection since they have no stable identity to collapse on. The same literature motivates emitting a leave event only when a user's last connection closes, rather than on every tab close, which this project implements by checking whether any of the user's connections remain before broadcasting a departure.

The use of a short time-to-live on presence records is also well supported in the literature on distributed presence, where it serves as a backstop against ghost entries left behind when a disconnect signal is missed. This project applies a twenty-four-hour expiry to its presence hashes for this reason, so that even a pathological missed cleanup cannot leave a stale participant visible indefinitely.

2.3 Shared Viewing, Snapshots, and Incremental Updates

Collaborative editing literature distinguishes broadly between two strategies for keeping remote views in sync with a source of truth: transmitting periodic full snapshots of the shared state, and transmitting incremental operations that describe individual changes.

Full snapshots are simple and self-healing any lost or misapplied update is corrected by the next snapshot, because each snapshot fully specifies the state at the cost of sending more data per update. Incremental operations are bandwidth-efficient but require careful handling of ordering and, when multiple editors are involved, of conflict resolution. The Live Shared Viewing phase described in this report deliberately adopts the full-snapshot strategy, transmitting the complete schematic model XML on each debounced change, because the design goal of this phase is one editor and many viewers rather than concurrent multi-editor editing. Under that goal, the self-healing property of snapshots is worth far more than the bandwidth a delta protocol would save, and no conflict resolution is required at all.

The literature on debouncing high-frequency UI events directly informs the snapshot cadence. Broadcasting on every low-level change event during a continuous interaction such as dragging a component would generate a flood of near-identical messages; the standard debounce pattern defer the action until a short quiet interval has elapsed collapses such a burst into a single update. This project debounces snapshots by 300 ms, a value chosen to feel immediate to a watching viewer while ensuring that a drag or a rapid sequence of edits produces one snapshot rather than dozens.

Incremental-update literature nonetheless remains relevant as a forward-looking concern. The design documented here reserves message types for component-level operations (add, move, remove, property change) so that a future phase can layer low-latency incremental updates on top of the snapshot channel without structural change, using the snapshot as a periodic correcting ground truth a hybrid the collaborative-systems literature identifies as a pragmatic middle path between pure-snapshot and pure-delta designs.

2.4 Reconnection and Backoff

Any system built on long-lived connections must plan for those connections to drop through transient network loss, a server restart, or a device sleeping. Distributed-systems literature strongly recommends exponential backoff for reconnection attempts: rather than retrying immediately and repeatedly, which can overwhelm a recovering server with a thundering herd of reconnections, each failed attempt waits for a progressively longer interval up to a cap. This project applies exponential backoff to both WebSocket

connections, starting at a short base delay, doubling after each failure, and capping the

delay so that reconnection remains bounded, with the delay reset once a connection succeeds.

2.5 Delegated Authentication and OAuth 2.0

OAuth 2.0 is the widely deployed standard for delegated authorisation, and its authorisation-code flow in which a provider such as Google redirects the user back to the application with a short-lived code that the application exchanges server-side for tokensis the pattern used for Sign in with Google. Security literature on this flow stresses several invariants that directly shaped the hardening work in this report. First, the state parameter must be present and validated on the callback; it exists to bind the callback to the original request and defend against cross-site request forgery, and a callback missing state should never be treated as a legitimate completion of the flow. Second, the authorisation code is meaningless without the paired state, so the presence of both parameters is a precondition for attempting the token exchange a requirement the original code violated by proceeding when either was present, which this project corrected.

The literature on robust integration with external identity providers further emphasises that every step which depends on a remote system the token exchange, the subsequent user-info request can fail independently, and that each must be guarded so that a failure produces a controlled, user-visible outcome rather than an unhandled server error. This directly motivated wrapping the token exchange, the user-info retrieval, and the local user/token persistence in separate guarded blocks, each redirecting cleanly to the login page with a specific message on failure.

2.6 Idempotency and Safe Account Provisioning

Account-provisioning literature recommends that the ensure a user exists for this verified identity step be idempotent: signing in twice with the same Google identity must converge on a single account rather than creating duplicates. The get-or-create pattern look up by the verifying attribute (here, the email returned by Google) and create only if absent the standard realisation of this, and is what the hardened flow uses for both the user record and its authentication token. The same body of work notes that an identity verified by a trusted external provider can legitimately be used to resolve a locally unverified account into a verified one, since the external provider has supplied exactly the

proof of ownership that local email verification would otherwise establish; this justifies the design decision to activate a previously inactive account upon a successful Google sign-in.

2.7 Automated Testing of Authentication Flows

Testing literature for security-sensitive flows recommends exercising the flow's failure branches at least as thoroughly as its success branch, because it is the failure branches missing parameters, provider errors, malformed responses that are least likely to be hit during casual manual testing and most likely to harbour latent defects. The recommended technique is to drive the view under test directly with crafted requests and to mock the external provider, so that provider-side outcomes (a token-exchange exception, a user-info response lacking an email) can be simulated deterministically without any real network call. This project's test suite follows this technique precisely: the OAuth callback view is invoked with a request factory, the OAuth session object is mocked, and each success and failure branch is asserted independently, as detailed in Chapter 4.

2.8 Configuration and Secrets Management

Finally, configuration-management literature for backend services recommends that secrets such as OAuth client credentials be sourced from the deployment environment rather than embedded in code, and that the service fail clearly and early when a required secret is absent rather than proceeding into an inevitable downstream error. The hardening work applies this by explicitly checking for empty client credentials at the top of the callback and returning an informative configuration error before any token exchange is attempted turning what had been an opaque deep failure into a clear, actionable one.

A theme common to both halves of this survey is that robustness lives at the boundaries: presence and viewing are exercises in gracefully handling connections that come and go, and authentication hardening is an exercise in gracefully handling every way an external sign-in can fall short of the happy path. The design decisions in the following chapters were shaped throughout by treating those boundary cases as first-class, not as afterthoughts.

Chapter 3

Problem Statement

3.1 Problem Statement

Before this fellowship's contribution, eSim on Cloud offered no way for two people to share a live view of the same schematic, and its Google sign-in path though usable in the ordinary case handled its own failure cases poorly. These are two separate problems, but each reduces to the same underlying difficulty: correctly managing a boundary between components that can behave imperfectly.

The collaboration problem

The schematic editor was fundamentally single-user in experience. When two learners opened the same shared project URL, the platform did nothing to make either aware of the other. There was no online indicator, no participant list, and no mechanism by which one person's edits could appear on another person's screen without a full manual save-and-reload. For a platform whose purpose includes classroom and peer-learning use, this absence was a real limitation: an instructor could not demonstrate a circuit live to a group, and two learners could not jointly inspect a design while discussing it.

Building live collaboration on top of an existing, single-user editor introduces several specific difficulties. The editor's graph is managed by a library (mxGraph) that was never designed with multi-user synchronisation in mind, so any synchronisation layer must sit beside it without corrupting its internal state or interfering with the existing save and simulation paths. Presence and live content are two different concerns with different lifetimes and different message rates, and conflating them risks making both fragile. A user may open the same project in several tabs, and the participant list must reflect people, not tabs. Anonymous guests, who have no stable identity, must still be counted sensibly. A viewer must be prevented from editing a schematic they do not own, yet must still be free to pan and zoom to inspect it. Snapshots must not cause the editor to re-render its own changes,

must not reset a viewer's chosen zoom and pan, and must not flood the connection during a drag. And any of the connections involved may drop at any moment and must recover on their own. Every one of these difficulties had to be solved without changing the behaviour that a learner working alone already relied upon.

The authentication problem

The Google sign-in path contained a set of latent defects that were invisible on the happy path but caused opaque failures the moment anything deviated from it. The callback attempted a token exchange whenever either the state or the code parameter was present, rather than requiring both, so a partial or malformed callback would proceed into a token exchange that then failed with an unhandled exception. There was no check for missing or empty OAuth credentials, so a deployment without configured credentials produced a confusing deep failure rather than a clear not configured message. The externally dependent steps token exchange and user-info retrieval were not individually guarded, so a provider-side error surfaced as a raw server error to the user. The token-issuing strategy depended on a module that had been removed in modern Django and, worse, created a token against an unfilled placeholder rather than the real user, meaning the token path was effectively broken. An account created through the sign-up form but never email-verified had no route to activation even after the same person proved ownership of the address by signing in with Google. On the browser side, a failed callback could leave a stale, invalid token in local storage. And none of this behaviour was covered by automated tests, so every one of these defects could only be found and could only be confirmed fixed by manual sign-in attempts.

The unifying requirement

Stated concisely, the problem this fellowship needed to solve was to make eSim on Cloud behave correctly at two boundaries it previously handled poorly: the boundary between multiple users sharing one project, and the boundary between the platform and an external identity provider. In both cases the requirement was not merely to make the happy path work it largely already did but to make the system behave predictably and recoverably when a connection drops, a tab closes, a credential is missing, a callback is

partial, or a provider call fails. Every design decision in the next chapter follows from that requirement.

3.2 Approach

3.2.1 Two Isolated WebSocket Channels for Collaboration

The collaboration subsystem is built as two deliberately separate WebSocket connections per client, established when a saved schematic loads: a presence channel at /ws/presence/<save_id>/ and a synchronisation channel at /ws/sync/<save_id>/. Each is handled by its own server-side consumer (PresenceConsumer and SyncConsumer) operating on its own channel group (presence_<save_id> and sync_<save_id>). The two groups never cross-post, and the synchronisation consumer's connect handler has no presence side-effects. This isolation is the single most important structural decision in the subsystem: it lets presence and live viewing be developed, tested, reasoned about, and fail independently. Figure 3.1 shows the resulting architecture.

3.2.2 Client-Computed Roles

Rather than fetching a role from the server, each client computes its own role from data it already has: it is the editor if the authenticated username equals the saved project's owner, and a viewer otherwise. This keeps role determination trivial and avoids an extra round trip, and it is applied directly to the local mxGraph instance by enabling or disabling editing. The owner receives a fully editable canvas; everyone else is placed in a constrained viewer mode.

3.2.3 Presence State in Redis

Presence membership is tracked in a Redis-backed PresenceStore holding two hashes per project: one mapping each live connection to a user key, and one mapping each user key to the user's displayed information. Authenticated users collapse to a single stable user key across tabs; anonymous guests are keyed per connection. A departure event is broadcast only when a user's last remaining connection closes. Both hashes carry a twenty-four-hour safety expiry.

3.2.4 Snapshot Relay with Debounce and Sender Exclusion

Live viewing transmits the full schematic model XML. The editor debounces outgoing snapshots by 300 ms so a burst of edits yields one message; the server relays each snapshot to the sync group while excluding the originating connection, so the editor never re-renders its own change; and viewers apply the snapshot through the existing gallery-render routine, which preserves their own zoom and pan. Either connection reconnects with exponential backoff on an unexpected drop.

3.2.5 Guarded, Idempotent OAuth Callback

For authentication, the approach is a single, carefully guarded callback view. The callback first requires both state and code; then checks for configured credentials; then performs the token exchange, the user-info retrieval, and the user/token persistence each inside its own guarded block, with any failure rendering the callback template in an error mode that clears any stale browser token and redirects cleanly to the login page. User provisioning is idempotent via get-or-create on the Google-supplied email, and a successful sign-in activates a previously inactive account. The token strategy is repaired to get-or-create a real token for the real user. Figure 3.2 shows the resulting decision flow.

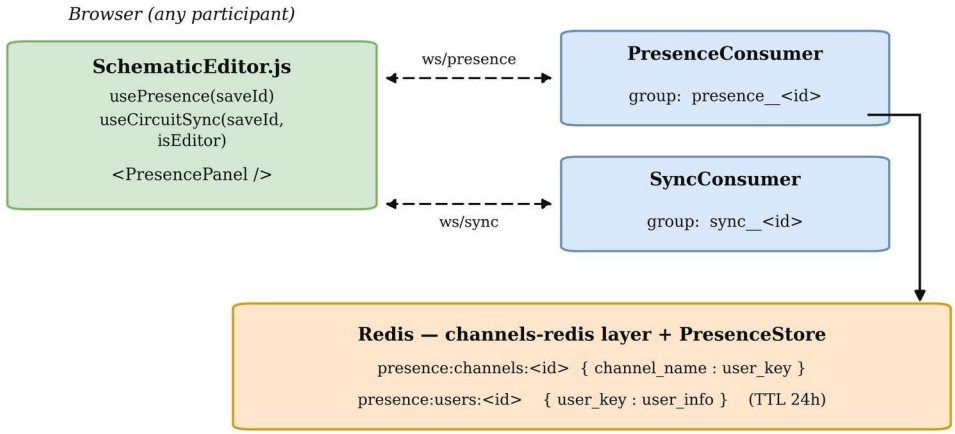

Figure 3.1: Live Collaboration architecture. Each browser holds two isolated WebSocket connections. Presence membership is tracked in a Redis-backed PresenceStore; both consumers use the channels-redis layer for group fan-out. Role (isEditor) is computed on the client from the project owner and the authenticated identity, not fetched.

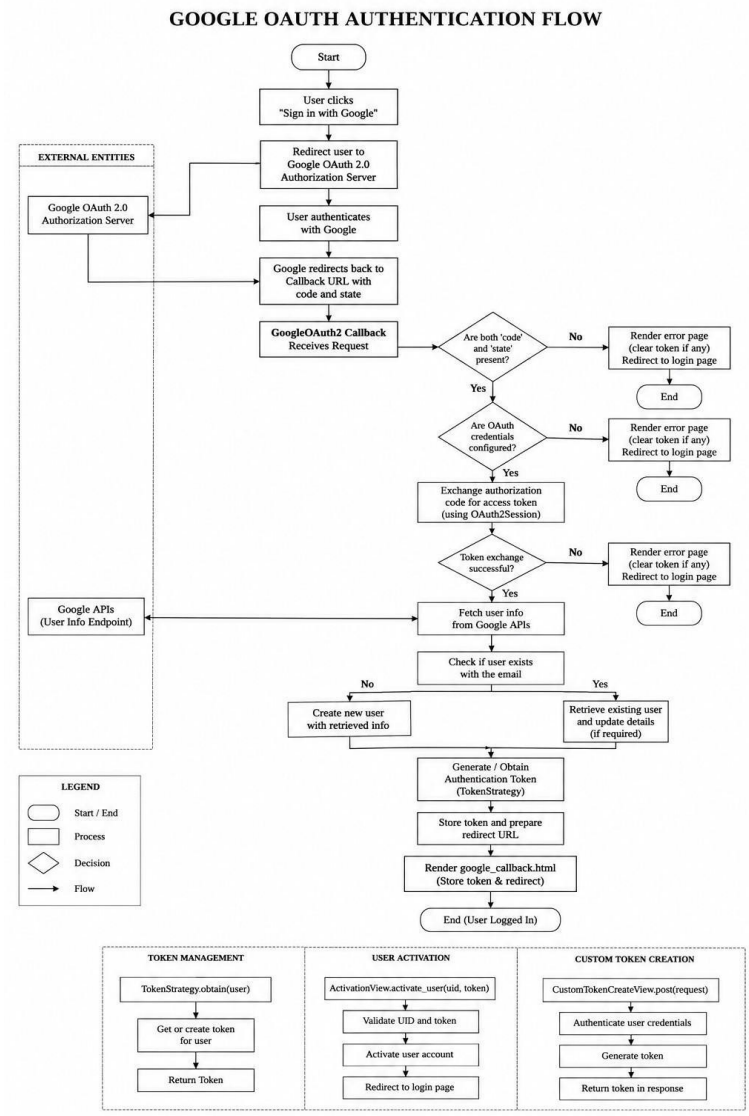

Figure 3.2: Hardened Google OAuth authentication and callback decision flow. Each externally dependent step is individually guarded; any failure converges on a single clean error path that clears any stale browser token and returns the user to the login page. The success path is idempotent and activates a previously inactive account, and the token-management, user-activation, and custom-token-creation paths are shown alongside.

3.2.6 Execution Stages

From an execution standpoint the combined approach has three stages. Stage one is entry: a learner signs in through the hardened OAuth path (or as an anonymous guest) and reaches the editor. Stage two is room formation: opening a shared project URL establishes the presence and sync WebSockets, the participant computes its role, and the presence store records its membership. Stage three is live session: presence events keep the participant list accurate while debounced snapshots keep every viewer's canvas current, with both channels

self-healing on disconnect. The next chapter documents the implementation of each stage in detail, together with the exact test cases and inputs used to validate it.

Chapter 4

Implementation

This chapter documents the implementation of the two contributions in two parts. Case 1 covers the Live Collaboration subsystem: its two-channel architecture, the presence layer (Phase 1), and the live shared viewing layer (Phase 2), together with the event catalog, the lifecycle of a room, and the exact validation scenarios each with its inputs used to confirm correct behaviour. Case 2 covers the Google OAuth authentication hardening: the corrected callback control flow, structured error handling, the repaired token strategy, the client-side recovery, and the dedicated automated test suite, again with every test case's inputs and expected outputs tabulated.

Throughout, particular attention is paid to which concrete scenario, with which inputs, produced which observed result, since the correctness of both subsystems is a claim about behaviour at boundaries that only specific, reproducible scenarios can substantiate.

4.1 Case 1: Live Collaboration Subsystem

Case 1 is delivered in two phases. Phase 1 (Presence & Project Rooms) makes participants aware of one another in real time. Phase 2 (Live Shared Viewing) streams the owner's schematic to all other participants. Both phases run over the two isolated WebSocket channels introduced in Chapter 3.

4.1.1 Files Added and Modified

The subsystem was implemented as a new Django app, collaborationAPI, on the backend, and a set of hooks, Redux slices, and components on the React front end. Table 4.1 lists the Phase 1 additions and the files modified to wire them in; Table 4.2 lists the Phase 2 additions on top.

Table 4.1: Files added and modified for Phase 1 (Presence & Project Rooms).

File	Purpose
Added backend	
collaborationAPI/consumers.py	PresenceConsumer (Phase 1)
collaborationAPI/presence.py	PresenceStore Redis-backed room state
collaborationAPI/auth.py	get_user_from_token() token → User
collaborationAPI/routing.py	WebSocket URL patterns
collaborationAPI/apps.py, _init_.py	App package & config
Added frontend	
utils/usePresence.js	React hook managing the presence WebSocket lifecycle
redux/reducers/presenceReducer.js	Redux slice for presence state
redux/actions/ collaborationActions.js	Action creators for all PRESENCE_* types
components/.../PresencePanel.js	Online list, join/leave toasts, role badge
Modified	
esimCloud/asgi.py	Register collaboration routing; add WS protocol handler
esimCloud/settings.py	Add collaborationAPI; configure channel layers + REDIS_URL
redux/actions/actions.js	Added 6 PRESENCE_* constants
redux/reducers/index.js	Registered presenceReducer
pages/SchematicEditor.js	Added usePresence(saveId) call

Table 4.2: Files added and modified for Phase 2 (Live Shared Viewing).

File	Change
utils/useCircuitSync.js	Added sync WebSocket hook: broadcasts snapshots (editor) or applies them (viewer)
collaborationAPI/consumers.py	Added SyncConsumer class
collaborationAPI/routing.py	Added /ws/sync/<save_id>/ route
.../Helper/ToolbarTools.js	Exported registerChangeListener, setViewerMode
.../Helper/SideBar.js	Added if (!graph.isEnabled()) return guard in drop handler
pages/SchematicEditor.js	Added owner/auth selectors, isEditor computation, useCircuitSync call
.../PresencePanel.js	Added Editing/Viewing role Chip badge

File	Change
.../Header.js	Fixed share URL (protocol separator; added version+branch params)

4.1.2 WebSocket Event Catalog

The two channels carry a small, closed set of message types, summarised in Table 4.3. Keeping the vocabulary small was a deliberate choice: it makes both consumers easy to reason about and leaves clearly reserved slots for a future incremental-update phase.

Table 4.3: WebSocket event catalog. The user shape is {username, is_anonymous}; xml is the full mxGraphModel string identical to the saved data_dump.

Channel	Dir	Type / payload	Notes
presence	C→S	PING	keepalive, every 25 s
presence	S→C	PONG	reply to PING
presence	S→C	USER_JOINED {user, users[]}	broadcast on connect
presence	S→C	USER_LEFT {user, users[]}	only on last-tab close
sync	C→S	PING / PONG	keepalive
sync	C→S	PROJECT_SNAPSHOT {xml}	editor only; debounced 300 ms
sync	S→C	PROJECT_SNAPSHOT {xml}	relayed to all except sender
sync	Reserved (Phase 3): COMPONENT_ADDED, COMPONENT_MOVED, COMPONENT_REMOVED, PROPERTY_CHANGED		

4.1.3 Phase 1 Presence Consumer and Store

Figure 4.1 gives the end-to-end Phase 1 picture: how a browser opening a shared project establishes a presence connection, how the consumer records membership through the channel layer and the Redis-backed PresenceStore, and how USER_JOINED and USER_LEFT events propagate to every other client, together with the high-level component and class breakdown that realises it.

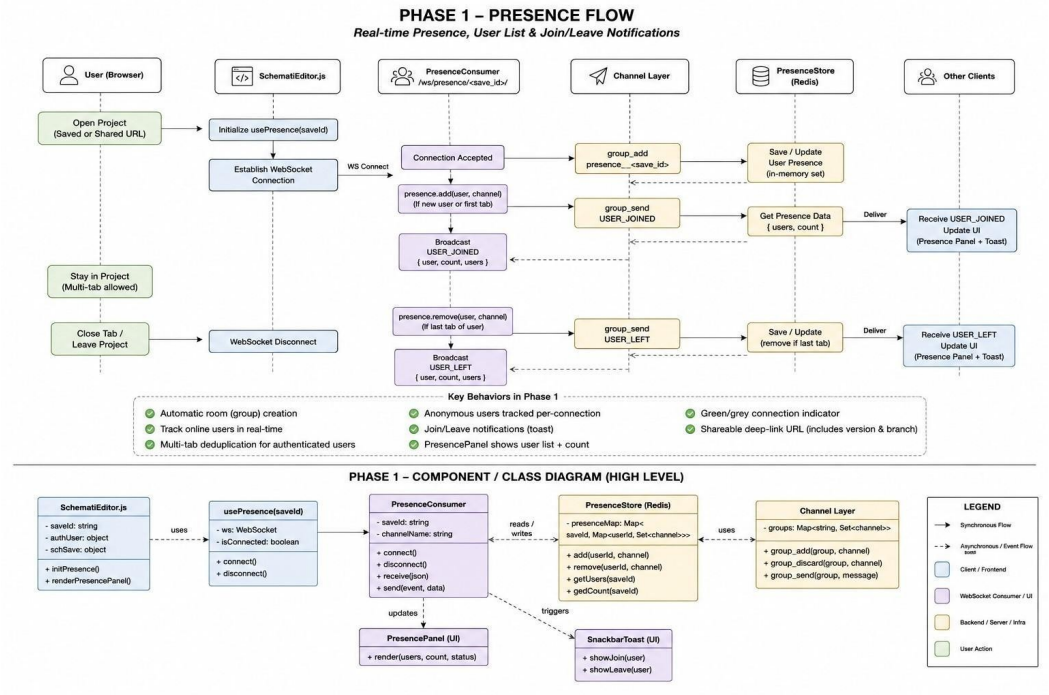

Figure 4.1: Phase 1 presence flow. Real-time presence tracking with automatic per-project rooms and join/leave notifications, together with the high-level component and class diagram.

On connect, the presence consumer resolves the connecting user from the token supplied on the WebSocket query, computes a stable user key (user_<pk> for an authenticated user, anon_<channel> for a guest), records the membership in the Redis-backed PresenceStore, and broadcasts a join to the room. Listing 4.1 shows the essential shape of the connect and disconnect handlers.

Listing 4.1: Presence consumer connect/disconnect (essential logic)

```
    class PresenceConsumer(AsyncWebsocketConsumer):
    async def connect(self):
    self.save_id = self.scope['url_route']['kwargs']
  ['save_id
 '] 4        self.group = f"presence_{self.save_id}" user
   5        = await get_user_from_token(self.scope)       # auth.py
   6
   7                if user and user.is_authenticated:
   8            self.user_key = f"user_{user.pk}"
   9                    self.user_info = {"username":
"is_anonymous": False}
10          else:
11                      self.user_key = f"anon_
12              {self.channel name}" self.user info =
"is_anonymous": True}
13
14          await self.channel_layer.group_add(self.group,
self.channel_name)
15    await self.accept()
16
```

```
17          PresenceStore.add(self.save_id, self.channel_name,
18
19          users = PresenceStore.get_users(self.save_id)  # de-
duplicated
20                  await
21              self.channel_layer.group_send(self.group,
22              { "type": "presence.event", "event":
23              "USER_JOINED", "user": self.user_info, "users":
24
   async def disconnect(self, code):
   last_tab = PresenceStore.remove(self.save_id,
  self.channel_name,
   self.user_key)
   await self.channel_layer.group_discard(self.group,
 self.channel_name)
29          if last_tab:  # broadcast leave only when the user's last
tab closed
30              users = PresenceStore.get_users(self.save_id)
31              await self.channel_layer.group_send(self.group, {
32                          "type": "presence.event", "event":
33                  "USER_LEFT", "user": self.user_info, "users":
34                  users
```

The store's remove returns True only when no other connection for that user key remains, which is exactly the condition under which a leave should be broadcast. On any Redis error the store methods log and return conservative defaults (remove returns True, get_users returns an empty list), so a transient Redis fault degrades the presence panel to empty rather than crashing the editor.

Presence lifecycle. Figure 4.2 traces the full presence lifecycle, including the multi-tab case that motivated the user-keyed store.

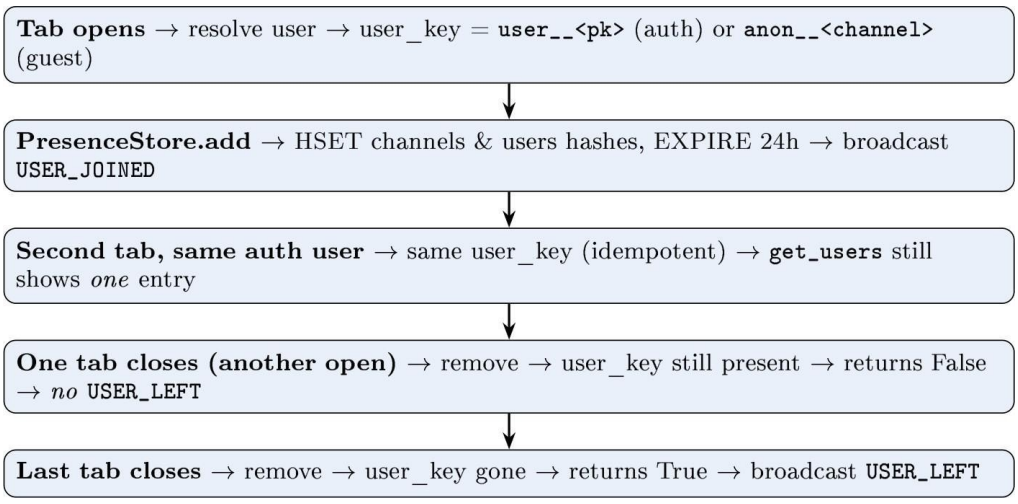

Figure 4.2: Presence lifecycle, including multi-tab de-duplication. A leave is broadcast only when a user's last connection closes.

4.1.4 Phase 1 Validation Matrix

Each Phase 1 behaviour was validated against a specific, reproducible scenario. Table 4.4 lists every check, the concrete inputs used to exercise it, the expected result, and the observed status. The inputs column records exactly the configuration under which the output was obtained for example, how many tabs, authenticated versus guest, and which action was taken so that each row is independently reproducible.

Table 4.4: Phase 1 validation matrix. Each row records the exact scenario and inputs that produced the observed status. Fixed marks checks that failed on an earlier build and passed after the share-URL repair in Header.js.

ID	Check	Inputs / scenario	Status
P1-1	/ws/presence/ <save_id>/ connects on schematic load	Open saved project ? id=S1&version=v1&branch=master; observe WS handshake 101 in dev tools	Pass
P1-2	USER_JOINED on opening shared URL	Tab A already in room; Tab B opens same URL as user bob; assert Tab A receives USER_JOINED{bob}	Pass
P1-3	USER_LEFT on tab close	Two users present; close bob's only tab; assert remaining user receives USER_LEFT{bob}	Pass
P1-4	Multi-tab dedup (auth user counted once)	User alice opens same project in 3 tabs; assert online count = 1 for alice, user list has one alice entry	Pass
P1-5	Anonymous user tracked per-connection	Two guest tabs (no token); assert online count = 2, both shown as Guest	Pass
P1-6	PresencePanel shows count + list	1 owner + 2 viewers present; assert panel shows 3 online and three named rows	Pass
P1-7	Join/leave toast auto-dismisses at 4 s	Trigger a join; assert Snackbar appears and disappears after 4 s	Pass
P1-8	Green/grey dot reflects connection state	Force WS close via dev tools; assert indicator turns grey; on reconnect turns green	Pass
P1-9	Share button generates valid deep-link URL	Click Share; assert clipboard URL has correct protocol separator (https://)	Pass, fixed
P1-10	Deep-link includes version+branch → fetch succeeds	Open shared URL in fresh browser; assert fetchSchematic loads the exact schematic	Pass, fixed

P1-11	Presence panel hidden when no schematic loaded	Navigate to editor without ?id=; assert panel not rendered	Pass

Two Phase 1 checks (P1-9 and P1-10) initially failed and are worth expanding, since they illustrate the boundary-first testing method. The share button had been producing a URL with a malformed protocol separator and, critically, omitting the version and branch query parameters. The observable symptom, with inputs open project S1 at version=v1, branch=master, click Share, paste the copied URL into a fresh browser tab, was that the second tab failed to load the schematic at all because fetchSchematic could not resolve which version to fetch. The repair in Header.js corrected the protocol separator and appended both parameters using window.location at render time; re-running the identical scenario then loaded the exact schematic and placed the second user into the same presence room, turning both P1-9 and P1-10 green.

4.1.5 Phase 2 Sync Consumer and Snapshot Relay

Figure 4.3 gives the end-to-end Phase 2 picture: how the editor's debounced canvas changes are serialised and sent as a PROJECT_SNAPSHOT, how the sync consumer relays that snapshot to the group while excluding the sender, and how each viewer renders it without losing its own zoom and pan, together with the high-level component and class breakdown.

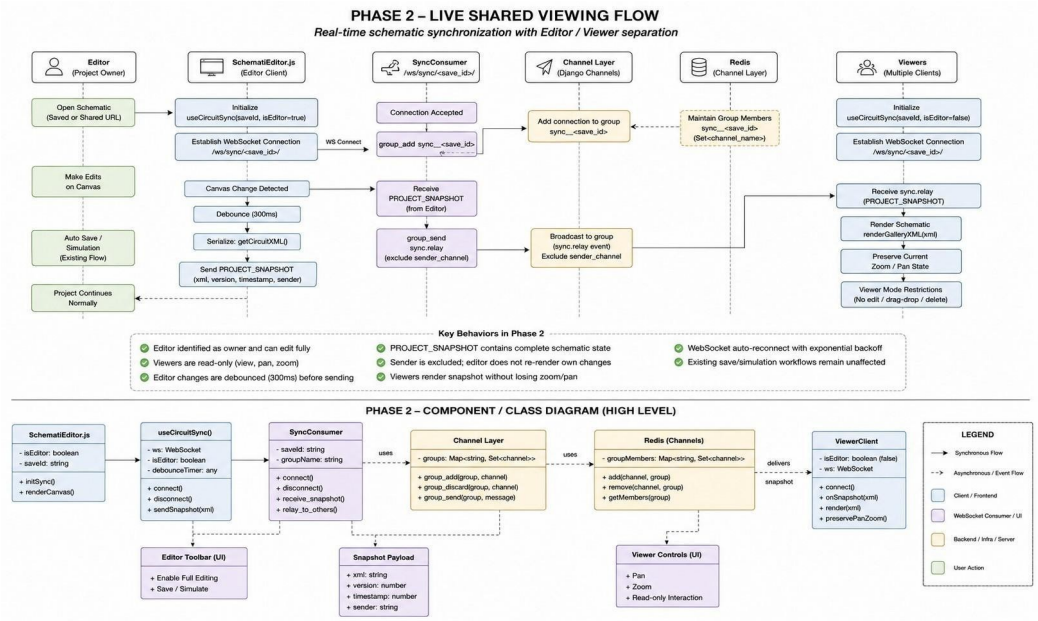

Figure 4.3: Phase 2 live shared viewing flow. Real-time schematic synchronisation with editor/viewer separation, together with the high-level component and class diagram.

Phase 2 adds the SyncConsumer. Its connect handler only joins the sync group no presence side-effects and its receive handler validates an incoming snapshot and relays it to the group with the sender's channel name attached, so the relay handler can skip the originator. Listing 4.2 shows the relay logic.

Listing 4.2: Sync consumer: receive, relay, and sender exclusion

```
      class SyncConsumer(AsyncWebsocketConsumer):
      async def connect(self):
      self.save_id = self.scope['url_route']['kwargs']
  ['save_id']
      self.room_group = f"sync_{self.save_id}"
      await self.channel_layer.group_add(self.room_group,
  self.channel_name)
          await self.accept()  # no presence side-effects

              async def receive(self,
          text_data): msg =
          json.loads(text_data)
                          if msg.get('type') ==
              'PROJECT_SNAPSHOT': xml =
              msg.get('xml')
self.channel_layer.group_send(self.room_group, {
                      'type': 'sync.relay', 'event_type':
'PROJECT_SNAPSHOT',
                      'payload': {'xml': xml},
                      'sender_channel': self.channel_name,  # for
exclusion
                  })

      async def sync_relay(self, event):
                  if self.channel_name ==
              return  # skip the editor itself
          await self.send(text_data=json.dumps({
                      'type': event['event_type'],
              **event['payload'],
```

On the front end, useCircuitSync attaches a change listener to the graph only for the editor, debounces by 300 ms, and sends the encoded XML; viewers instead receive snapshots and re-render. Crucially, the listener is attached after the initial fetchSchematic/render, so the first load never produces a spurious broadcast. Listing 4.3 shows the editor and viewer branches.

Listing 4.3: useCircuitSync: editor debounce-broadcast and viewer apply

```
// Editor branch: debounce graph changes, then broadcast the full
XML
     if (isEditor)
   { registerChangeListener(() => {
              clearTimeout(debounceRef.current
     ); debounceRef.current = setTimeout(()
       const xml = Save();  // encode mxGraphModel -> XML
               ws.send(JSON.stringify({ type: 'PROJECT_SNAPSHOT',
       xml }));
         }, 300);
 }
```

```
11
12 // Viewer branch: apply incoming snapshot without touching
zoom/pan
13  ws.onmessage = (e) => {
14        const msg = JSON.parse(e.data);
15        if (msg.type === 'PROJECT_SNAPSHOT' && !
16      renderGalleryXML(msg.xml);  // clears child cells + redraws;
scale/translate kept
17    }
18 };
```

Viewer mode enforcement. When a participant is not the owner, setViewerMode(true) calls graph.setEnabled(false) (blocking selection, move, connect, and delete) while graph.setPanning(true) keeps pan and zoom available; the sidebar drop handler additionally returns early when !graph.isEnabled(), blocking component drops. One pre-existing gap was noted and documented rather than silently worked around: a keyboard-shortcut guard elsewhere reads graph.isEnabled as a property rather than calling it, so an Alt+Rotate shortcut can still fire in viewer mode. This is a pre-existing defect not introduced by this work and is recorded as an accepted Phase 2 limitation.

Snapshot synchronisation lifecycle. Figure 4.4 summarises the full edit-to-view cycle, including the two properties that make it correct: sender exclusion and zoom/pan preservation.

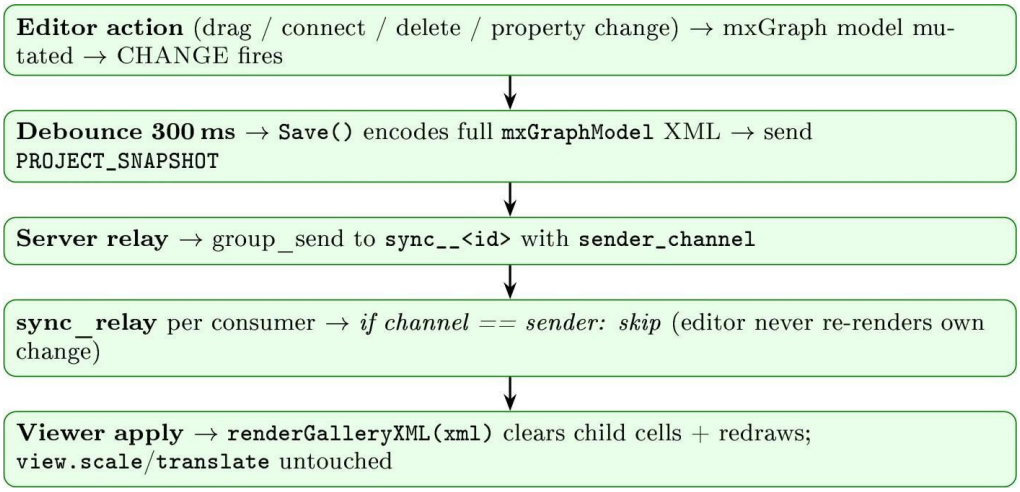

Figure 4.4: Snapshot synchronisation lifecycle. Sender exclusion prevents the editor from re-rendering its own change; applying the snapshot preserves each viewer's independent zoom and pan.

4.1.6 Phase 2 Validation Matrix

Table 4.5 lists the Phase 2 checks with their concrete inputs. The scenarios use a canonical two-role setup unless stated otherwise: an owner alice (editor) and a viewer bob, both on project S1.

Table 4.5: Phase 2 validation matrix. Fixed marks checks (S2-2, S2-3) that previously misbehaved every participant showed Viewing and the owner's canvas was not editable until the isEditor computation was corrected in SchematicEditor.js.

ID	Check	Inputs / scenario	Status
S2-1	/ws/sync/ connects on load	Open S1; assert second WS handshake 101 alongside presence	Pass
S2-2	Owner chip = Editing, others = Viewing	alice owns S1; assert alice's chip “Editing”, bob's “Viewing”	Pass, fixed
S2-3	Owner canvas fully editable	As alice: drag, connect, delete a component; assert all succeed	Pass, fixed
S2-4	Viewer canvas: move/delete/connect blocked	As bob: attempt move, delete, connect; assert all no-op	Pass
S2-5	Viewer canvas: sidebar drag-drop blocked	As bob: drag a component from sidebar onto canvas; assert rejected	Pass
S2-6	Viewer canvas: pan + zoom work	As bob: pan by drag, zoom via toolbar; assert both succeed	Pass
S2-7	Editor change → 300 ms debounce → snapshot sent	As alice: move a component; assert exactly one PROJECT_SNAPSHOT frame sent after 300 ms quiet	Pass
S2-8	Sender excluded (editor doesn't re-render own canvas)	As alice: edit; assert alice's onmessage receives no snapshot for her own edit	Pass
S2-9	Viewer receives snapshot; renderGalleryXML called	alice adds a resistor; assert bob's canvas shows the new resistor within one debounce cycle	Pass
S2-10	Viewer zoom/pan preserved across rerenders	bob zooms to 200% and pans; alice edits; assert bob's zoom/pan unchanged after apply	Pass
S2-11	WS reconnects with exponential backoff on drop	Kill sync WS via dev tools; assert reconnect attempts at ≈2, 4, 8 . . . s capped at 30 s; delay resets	Pass

		on success	
S2-12	Editing persists normally (save/simulation unaffected)	As alice: edit, then Save and Simulate; assert both behave exactly as pre-collaboration	Pass

Worked example for S2-9 and S2-10. The most instructive Phase 2 test combines content synchronisation with view preservation, because it is where the design's two subtlest properties meet. Inputs: owner alice and viewer bob both on project S1; bob first sets zoom to 200% and pans the canvas so that the bottom-right of the schematic is centred; alice then drags a resistor R1 from the sidebar onto the canvas at grid position roughly (240, 180) and pauses. Expected output: after one 300 ms debounce, bob's canvas shows the new R1 at the same logical position, and bob's zoom remains 200% with the same pan offset the new component simply appears within his existing view rather than the view snapping back to a default scale. Observed output: exactly this; renderGalleryXML cleared and redrew the child cells, adding R1, while leaving graph.view.scale and graph.view.translate untouched, so bob's independent inspection viewport was preserved across the update. This confirmed both S2-9 (content propagates) and S2-10 (view is not disturbed) in a single scenario.

Worked example for S2-11 (reconnection). Inputs: with a live owner/viewer session established, the sync WebSocket is force-closed from the browser's network panel to simulate a transient drop while the presence socket is left untouched. Expected output: the sync hook's onclose fires and schedules a reconnect after ≈2 s; if that attempt also fails, the delay doubles to 4 s, then 8 s, and so on, capped at 30 s; on a successful reconnect the backoff delay resets to its 2 s base, and the next editor snapshot repopulates the viewer. Observed output: reconnection attempts followed the doubling schedule and the viewer resumed receiving snapshots after the socket recovered, with the presence panel's connection dot tracking the transition grey→green. This validates that a dropped viewing session self-heals without any user action.

4.1.7 Room Lifecycle

Bringing the two phases together, Figure 4.5 traces the full life of a collaboration room from the moment a shared URL is opened to the moment the last participant leaves, showing where each of the two WebSocket connections is established, used, and torn down.

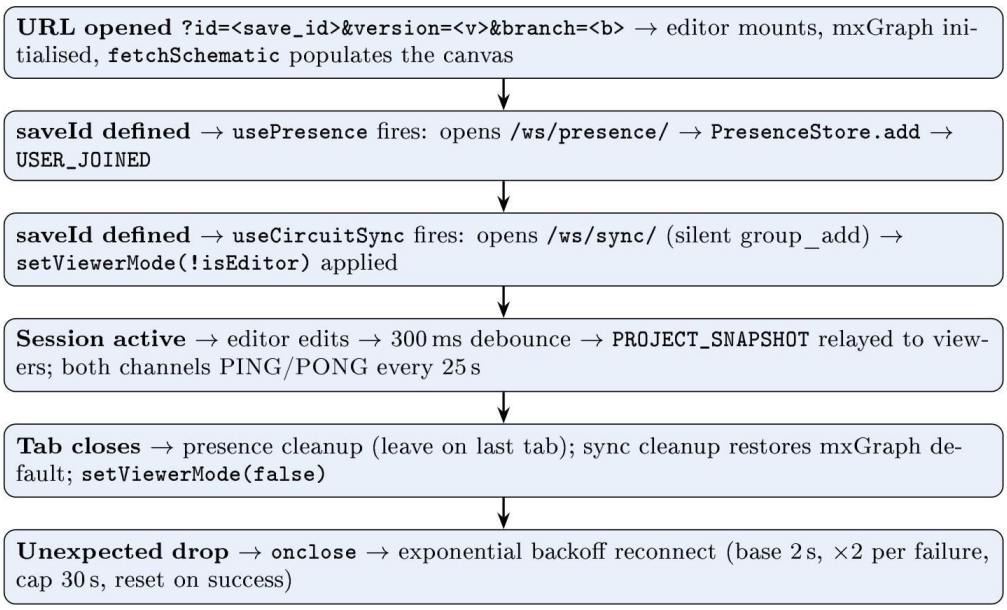

Figure 4.5: Room lifecycle across both channels, from opening a shared URL through active use to tab close and unexpected-drop recovery.

4.1.8 Configuration Reference

Following the configuration-management guidance from the Literature Survey, the subsystem's tunable constants are sourced from settings and the deployment environment rather than scattered as literals. Table 4.6 summarises them.

Table 4.6: Configurable constants governing the Live Collaboration subsystem.

Constant	Value	Purpose
REDIS_URL / channel layer	(env)	Backing store for the channels-redis layer and the PresenceStore
Presence hash TTL	24 h	Safety expiry on presence hashes; backstop against ghost entries
Snapshot debounce	300 ms	Collapses a burst of editor edits into one PROJECT_SNAPSHOT
Keepalive interval	25 s	PING cadence on each channel; PONG confirms liveness
Backoff base / cap	2 s / 30 s	Reconnection delay: doubles per failure, capped, reset on success
Toast auto-dismiss	4 s	Join/leave Snackbar lifetime in PresencePanel

4.1.9 Known Limitations and Technical Notes

Several limitations were identified during development and are recorded honestly here rather than silently worked around, so that a future contributor inherits an accurate picture. Table 4.7 lists them with severity.

Table 4.7: Known limitations and technical notes for the Live Collaboration subsystem. Most are pre-existing or trivial; none block correct Phase 1/2 behaviour.

Item	Severity	Note
graph is a module-level singleton	Pre-existing	registerChangeListener/setViewerMode depend on it; a hot reload could silently no-op. Inherited, not introduced here.
Keyboard-shortcut guard gap	Low	if (graph.isEnabled) (no parentheses) is always truthy, so Alt+Rotate still fires in viewer mode. Pre-existing; fix is graph.isEnabled().
isEditor computed in two places	Low	Editor page and presence panel each compute it; extract to a shared selector to prevent divergence.
No Redux slice for sync state	Low	useCircuitSync is self-contained; sync connection status is not observable elsewhere. Acceptable until a UI indicator is desired.
Independent keepalives	Trivial	Presence and sync each PING every 25 s two pings per tick. Harmless redundancy.
Share URL uses window.location at render	Trivial	Correct for the current /eda/ deployment path; would need revisiting for a non-root base path.

These notes double as a lightweight maintainer hand-off: none of the items affect the correctness demonstrated in the Phase 1 and Phase 2 validation matrices, but each is a natural candidate for a small future cleanup, and the keyboard-shortcut gap in particular is listed as a concrete fix in the Future Scope chapter.

4.2 Case 2: Google OAuth Authentication Hardening

Case 2 hardened the Sign in with Google path. The work touched four files: the callback view (authAPI/views.py), the token strategy (authAPI/token.py), the callback template (authAPI/templates/google_callback.html), and the front-end login error handler, plus a new, dedicated test module (authAPI/tests.py). The guiding principle was reproduce-then-repair: every defect was first captured as a failing test that drove the callback view directly with a crafted request and a mocked Google session, then fixed until the test passed.

Figure 4.6 shows the class model of the components involved: how the GoogleOAuth2 callback view relates to the User and Token models, to the OAuth2Session that performs the token exchange and user-info retrieval, and to the TokenStrategy, ActivationView, and CustomTokenCreateView that issue and manage tokens.

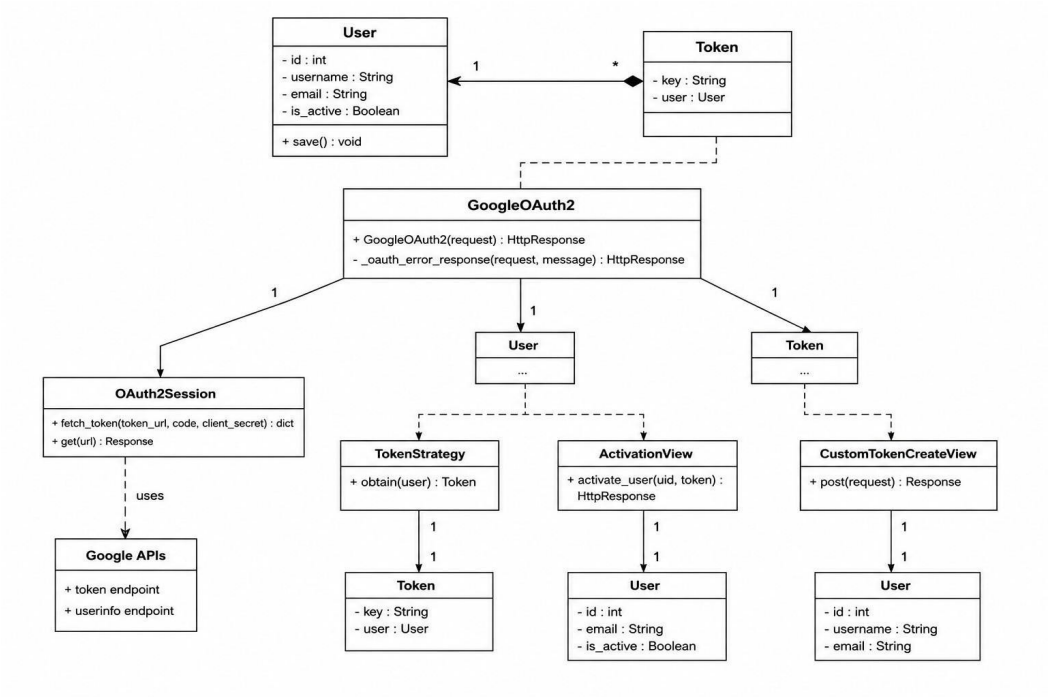

Figure 4.6: Class model of the hardened authentication components User and Token in relation to GoogleOAuth2, OAuth2Session, TokenStrategy, ActivationView, and CustomTokenCreateView.

4.2.1 Corrected Callback Control Flow

The most consequential defect was in the guard that decides whether to attempt a token exchange. The original condition proceeded when either state or code was present; because the authorisation-code flow is meaningless without both, a partial callback slipped through and crashed inside fetch_token with an unhandled exception. The repaired guard requires both parameters and returns a clean error otherwise. Listing 4.4 contrasts the two.

Listing 4.4: Callback parameter guard: original defect vs. repair

```
# BEFORE (defect): proceeds when EITHER param is present -> unhandled crash
# if not (state is None) or not (code is None):
#     ... attempt token exchange ...

# AFTER (repair): both are required; fail cleanly otherwise
if state is None or code is None:
            logger.warning("OAuth callback missing params: state=
```

```
 %s
",                                                    "pres
                    ent" if state else "missing",
                    "present" if code else "missing")
    return _oauth_error_response(request,
```

4.2.2 Fail-Safe Credential Check and Guarded Steps

Immediately after the parameter guard, the callback verifies that the OAuth client credentials are actually configured, returning a clear not configured message before any network call. Each externally dependent step token exchange, user-info retrieval, and user/token persistence is then wrapped in its own try/except, and a shared helper, _oauth_error_response, renders the callback template in error mode so the browser clears any stale token and redirects to login. Listing 4.5 shows the credential check and the shape of a guarded step.

Listing 4.5: Credential check and a guarded external step

```
    def _oauth_error_response(request, message):
    protocol = 'https://' if request.is_secure() else 'http://'
    web_url = protocol + request.get_host() + '/eda/#/login'
    return render(request, 'google_callback.html',
    {'token': None, 'url': web_url, 'error': message})

client_id = settings.SOCIAL_AUTH_GOOGLE_OAUTH2_KEY
client_secret = settings.SOCIAL_AUTH_GOOGLE_OAUTH2_SECRET
if not client_id or not client_secret:  # fail early & clearly
    logger.error("Google OAuth credentials not configured.")
    return _oauth_error_response(request,
                "Google OAuth is not configured on this

try:      # guarded token exchange
            google = OAuth2Session(client_id,
        redirect_uri=settings.GOOGLE_OAUTH_REDIRECT_URI,
        state=state)

google.fetch_token('https://accounts.google.com/o/oauth2/token',
   client_secret=client_secret, code=code)
   except Exception as exc:
   logger.error("Google token exchange failed: %s", exc)
   return _oauth_error_response(request,
   "Google authentication failed. Please try again.")
```

4.2.3 Idempotent Provisioning and Account Activation

Once a verified email is obtained, the user is provisioned idempotently via get-or-create on the email, and a previously inactive account is activated since a successful Google sign-in is sufficient proof of ownership. The email itself is validated (a user-info response lacking an email is rejected). Listing 4.6 shows the persistence block.

Listing 4.6: Idempotent user provisioning with activation of inactive accounts

```
        email = user_info.get('email')
        if not email:
        logger.error("Google user info missing the email field")
        return _oauth_error_response(request,
        "No email address was returned from Google.")
        try:
        user, created =
  get_user_model().objects.get_or_create(email=ema
  il)
        if created:
        user.username = user_info.get('name',
  'user').strip().replace(' ', '_') \
  10 + str(randint(0, 9999))
     user.is_active = True
     user.save()
     elif not user.is_active: # form sign-up, never verified
     user.is_active = True # Google-verified -> activate
     user.save()
     token, _ = Token.objects.get_or_create(user=user)
     except Exception as exc:
     logger.error("Failed to persist user/token for %s: %s",
  email, exc)
```

4.2.4 Repaired Token Strategy

The token strategy was independently broken: it imported a helper (django.utils.six) removed in modern Django, and it created a token against an unfilled placeholder (Token.objects.create(user=...) where ... is Python's Ellipsis) rather than the authenticated user. The repair removes the dead import and issues the token for the real user via get-or-create, as in Listing 4.7.

Listing 4.7: Repaired token strategy

```
        class TokenStrategy:
        @classmethod
        def obtain(cls, user):
        from rest_framework.authtoken.models import Token
        token, _ = Token.objects.get_or_create(user=user) # real
  user, idempotent
        return token
```

4.2.5 Callback Template and Client-Side Recovery

The callback template now branches on an error flag: on error it removes any stale token from local storage and redirects to login; otherwise it stores the issued token and redirects to the saved destination. The front-end login handler was likewise updated so that a failed Google sign-in surfaces a specific message rather than a generic one. Listing 4.8

shows the template branch.

Listing 4.8: Callback template: clear stale token on error, else store token

```
      // google_callback.html (inline script)
      // {% if error %}
      window.localStorage.removeItem('esim_token'); // do not keep
  an invalid token
4  window.open('{{ url }}', '_self');  // -> /eda/#/login
5  // {% else %}
6  window.localStorage.setItem('esim_token', '{{ token }}');
7  var redirect_to = window.localStorage.getItem('ard_redurl');
8  window.open(redirect_to || '{{ url }}', '_self');
9  // {% endif %}
```

4.2.6 Automated Test Suite

The dedicated test module drives the callback view directly with a RequestFactory and a mocked OAuth2Session, using override_settings to inject fake or empty credentials. This lets every success and failure branch be asserted deterministically, with no real network call. Table 4.8 tabulates each test with its exact inputs (parameters and mocked provider behaviour) and the asserted output. The fixed fake credentials used throughout are KEY=fake_client_id, SECRET=fake_client_secret; the fixed fake user-info is {email: testuser@example.com, name: Test User}.

Table 4.8: Google OAuth automated test suite. Each row records the exact request parameters and mocked provider behaviour (the inputs) and the asserted output, so every case is independently reproducible via python manage.py test authAPI.

Test	**Inputs (params / mocked provider)**	**Asserted output**
Parameter validation		
missing_state	GET ?code=somecode only	200; no esim_token in body
missing_code	GET ?state=somestate only	200; no esim_token
both_missing	GET with no params	200; no esim_token
Missing credentials		
both_empty	KEY=, SECRET=; state,code present	200; no esim_token; no network call

Test	**Inputs (params / mocked provider)**	**Asserted output**

key_only_missing	KEY=, SECRET=real	200; no esim_token
secret_only_missing	KEY=real, SECRET=	200; no esim_token
Token / user-info failure		
fetch_token_exception	fetch_token raises invalid_grant	200; no token; User.count()==0
user_info_exception	session.get raises network error	200; no token; User.count()==0
missing_email	user-info {name: No Email Here}	200; no esim_token
Successful flow		
new_user_created	valid params; user-info = fake user	200; user active; token exists
existing_active_not_dup	pre-existing active user, same email	200; user count for email = 1
inactive_user_activated	pre-existing is_active=False user	200; user becomes active
token_key_present	valid flow	token key appears in rendered body
Activation view & inactive login		
activation_renders	GET /activate/uid/token/	200; body has uid, token, activation URL
activation_no_slash	GET /activate/uid1/tok1 (no slash)	status in {200, 301}
inactive_login_error	POST token for is_active=False user	400; error says activated, not incorrect

Worked example: inactive_user_activated. This test captures the activation objective precisely. Inputs: a user is pre-created with email=testuser@example.com and is_active=False (modelling someone who signed up via the form but never verified their email); the mocked OAuth2Session.fetch_token returns {access_token: tok} and the mocked get().json() returns the fake user-info for that same email; the callback is invoked with ?state=somestate&code=somecode under the fake credentials. Expected output: the callback resolves the existing user by email (not a duplicate), sets is_active=True, issues a token, and renders the success template with HTTP 200. Observed output: after refresh_from_db(), user.is_active is True and the response is 200 confirming that a

Google-verified identity now rescues a previously dead-ended unverified account, which before this work had no path to activation.

Worked example: fetch_token_exception. Inputs: valid fake credentials and both parameters present, but the mocked fetch_token is configured to raise Exception("invalid_grant"), modelling a stale or replayed authorisation code. Expected output: the guarded token-exchange block catches the exception, returns the error-mode template with HTTP 200, and critically creates no user (User.objects.count()==0), so a failed exchange leaves no half-provisioned account behind. Observed output: exactly this, confirming both the clean error path and the no-orphan-user guarantee stated in Chapter 3.

4.2.7 Manual Cross-Check

Alongside the automated suite, the end-to-end flow was exercised manually in a browser against a development deployment: a first-time Google sign-in (new-user creation), a repeat sign-in with the same account (idempotent, no duplicate), a deliberately cancelled Google consent screen (clean return to login with a specific message), and a run with credentials removed from the environment (not configured message). Each manual scenario corresponds to one or more rows of Table 4.8, and agreement between the manual result and the automated assertion was treated as the standard of correctness.

Chapter 5

Conclusion and Future Scope

This fellowship delivered two working contributions to eSim on Cloud: a real-time Live Collaboration subsystem for the schematic editor, and a hardened Google OAuth authentication flow. Both were built to behave predictably at the boundaries where components meet and can misbehave, and both were validated against concrete, reproducible scenarios rather than by inspection alone.

The Live Collaboration subsystem introduced two isolated WebSocket channels per participant presence and synchronisation so that awareness of who is present and streaming of the owner's schematic are handled independently and can fail independently. Presence is tracked in a Redis-backed store keyed by a stable user identity, so multiple tabs of the same authenticated user collapse to a single visible participant while anonymous guests are tracked per connection, and a departure is announced only when a user's last connection closes. Live viewing transmits full schematic snapshots, debounced by 300 ms and relayed with the originating editor excluded, and applies them on each viewer's canvas without disturbing that viewer's own zoom and pan. Viewer mode blocks editing while preserving inspection, and both channels reconnect with exponential backoff. The Phase 1 and Phase 2 validation matrices confirmed correct behaviour across presence, multi-tab de-duplication, role separation, snapshot propagation, view preservation, and reconnection, with the two share-URL defects and the two role defects found and fixed during testing.

The authentication hardening turned a sign-in path that failed opaquely into one that fails clearly and recovers cleanly. The callback now requires both OAuth parameters before attempting a token exchange, checks for configured credentials before any network call, guards each external step so a provider failure produces a controlled redirect to login rather than a raw server error, provisions users idempotently, activates previously unverified accounts on a successful Google-verified sign-in, and repairs a token strategy that had been issuing tokens against a placeholder user. A dedicated automated test suite pins down every one of these behaviours parameter validation, missing credentials, token-exchange and

user-info failures, the new-user and returning-user success paths, inactive-account activation, and the inactive-login error message so that the sign-in flow is now regression-protected and reproducible without manual attempts.

5.1 Future Scope

Several concrete extensions follow naturally from this work.

Collaborative editing (Phase 3). The synchronisation channel and consumer were deliberately designed to accommodate incremental, component-level events COMPONENT_ADDED, COMPONENT_MOVED, COMPONENT_REMOVED, PROPERTY_CHANGED which are already reserved in the event catalog. A pragmatic Phase 3 would run these incremental events alongside the existing snapshot channel: the incremental events give low-latency visual feedback, while the periodic full snapshot remains the self-healing ground truth that corrects any drift, avoiding the need for full operational-transform machinery until genuine concurrent multi-editor editing is required. Only if simultaneous multi-editor editing becomes a goal would a sequence-numbered ordering or operational-transform layer be warranted.

Editor hand-off and multi-editor roles. The current model assigns the editor role to the project owner. A natural extension is an explicit hand-off, so the owner can grant temporary edit rights to a viewer useful in a teaching setting where an instructor wants a learner to make a change while the group watches. This builds directly on the existing client-computed role mechanism.

Cursor and selection presence. Beyond simple online awareness, showing each participant's cursor position or current selection would make collaboration feel substantially more live. Because presence already has its own channel and store, this is a natural incremental addition rather than a structural change.

Refactoring shared role logic. The isEditor computation currently appears in two places (the editor page and the presence panel), each independently selecting the owner and the authenticated identity. Extracting this into a single selector or context would remove a small but real duplication and ensure the two never diverge.

Fixing the documented viewer-mode gap. The pre-existing keyboard-shortcut guard that reads graph.isEnabled as a property rather than calling it should be corrected to graph.isEnabled() so that Alt+Rotate is properly blocked in viewer mode, closing the one accepted Phase 2 limitation.

Authentication: additional providers and CSRF-state validation. The hardened callback structure generalises cleanly to other identity providers. A further hardening step is to not merely require the state parameter but to validate it against a value stored at the start of the flow, fully realising its CSRF-protection purpose. The guarded, testable structure delivered here is the right foundation for that addition.

Broader test coverage and observability. The collaboration subsystem would benefit from automated integration tests that drive the two consumers directly simulating joins, leaves, multi-tab connections, and snapshot relays to match the regression protection the authentication suite already provides. Emitting structured lifecycle events (join, leave, snapshot relayed, reconnect) to a central logging or metrics system would also let future operators observe collaboration health without manual WebSocket inspection.

5.2 Closing Remarks

The most valuable outcome of this fellowship is that both contributions are largely invisible when they work: two learners simply find themselves in a shared, live room without configuring anything, and a learner simply signs in with Google and reaches the editor, or receives a clear message if something is wrong. This invisibility is exactly what the FOSSEE mission needs at classroom and MOOC scale, where neither learners nor instructors should have to think about WebSockets, Redis, presence de-duplication, or OAuth callbacks.

The experience reinforced several lessons that generalise beyond this project: that real-time features should be decomposed into independent channels with independent failure modes rather than a single coupled connection; that presence must be reasoned about in terms of distinct people, not distinct connections; that full snapshots are often worth their bandwidth for the self-healing simplicity they buy; that every externally dependent step deserves its own guard and its own clear error; and that reproduce-then-repair, backed by tests that pin down each failure branch, is the discipline that turns latent defects into fixed ones. These lessons transfer directly to the larger open-source infrastructure contributions

the author hopes to pursue beyond the fellowship period.

Chapter 6

Bibliography

[1] Django Software Foundation. Django Documentation. Consulted for the authentication views, template rendering, user model, and override_settings testing utilities used throughout the OAuth hardening work.

[2] Django Channels Project. Django Channels Documentation. Sections on ASGI, consumers, channel layers, and channel groups, consulted while implementing the presence and synchronisation consumers and their group-based fan-out.

[3] Redis Ltd. Redis Documentation. Sections on hashes, key expiry, and time-to-live, consulted while designing the Redis-backed PresenceStore and its multi-tab de-duplication.

[4] The IETF OAuth Working Group. The OAuth 2.0 Authorization Framework (RFC 6749) and OAuth 2.0 Security Best Current Practice. Consulted for the authorization-code flow, the role of the state parameter, and guidance on guarding each externally dependent step of the callback.

[5] Google Identity. Sign in with Google / OAuth 2.0 for Web Server Applications. Consulted for the token-exchange endpoint and user-info retrieval used in the callback.

[6] Mozilla Developer Network. WebSockets API Documentation. Consulted while implementing the two client WebSocket connections, their keepalive ping/pong, and the exponential-backoff reconnection logic.

[7] JGraph Ltd. mxGraph User Manual. Consulted for the graph model, change events, enabling/disabling interaction, panning, and view scale/translate semantics relied upon by viewer mode and by non-destructive snapshot application.

[8] React and Redux Documentation. Consulted for the custom hooks (usePresence, useCircuitSync) and the Redux slice used for presence state on the front end.

[9] Fielding, R. T. Architectural Styles and the Design of Network-based Software Architectures. Doctoral dissertation, 2000. Consulted for background on idempotency and resource-oriented design applied to user/token provisioning.

[10] Kleppmann, M. Designing Data-Intensive Applications. O'Reilly Media, 2017. Consulted for background on snapshot-versus-incremental synchronisation strategies and on reconciliation of externally held state.

[11] FOSSEE, Indian Institute of Technology Bombay. FOSSEE Fellowship and Internship Programme Materials, 2026. Used for reporting conventions and acknowledgement context.