Tuwunel Workers - Technical Deep Dive (Narrative)

Tuwunel Workers represents an ambitious implementation of the Matrix protocol—a federated, decentralized communication standard—built entirely on Cloudflare's edge computing infrastructure. Unlike traditional Matrix homeservers like Synapse or Dendrite that run on conventional server architectures, this project reimagines the entire homeserver stack as a globally distributed, serverless application leveraging Cloudflare Workers' unique capabilities. The system provides complete instant messaging functionality including end-to-end encrypted messaging, media sharing, real-time notifications, and the foundational elements for federation, all while operating within the constraints and advantages of edge computing.

The architectural foundation rests on five core Cloudflare services working in concert. Cloudflare Workers handle all HTTP request routing and API processing, serving as the stateless entry point for Matrix clients and federation partners. D1, Cloudflare's distributed SQLite database, provides persistent storage for all critical data including user accounts, room definitions, message events, room memberships, device information for end-to-end encryption, and push notification configurations. The system uses three separate KV namespaces for different caching and session management needs: SESSIONS stores access tokens and authentication state, DEVICE_KEYS maintains end-to-end encryption key material for quick lookup, and CACHE holds general-purpose cached data like room summaries and user profiles. R2 object storage handles all media uploads including images, videos, audio files, and documents, with plans to integrate Cloudflare's image resizing services for thumbnail generation. Finally, three types of Durable Objects—RoomDurableObject, SyncDurableObject, and FederationDurableObject—provide strongly-consistent coordination points for real-time features, ensuring that concurrent operations on the same room or user session maintain consistency without race conditions.

The authentication system implements Matrix's User-Interactive Authentication flow, a multi-stage authentication mechanism that supports various credential types. When a user registers a new account, the system generates a unique Matrix user ID in the format @username:servername, validates password strength requirements, and stores a bcrypt-hashed password in the D1 database. Login endpoints accept either username or Matrix ID along with the password, verify credentials against stored hashes, and generate a cryptographically random access token that gets stored in the SESSIONS KV namespace with the user ID and device ID as metadata. Every authenticated API request includes this access token in the Authorization header, which the auth middleware validates by looking up the token in KV, retrieving the associated user and device identifiers, and attaching them to the request context for downstream handlers. The system supports multiple simultaneous sessions across different devices, with each device receiving its own access token. Logout functionality allows terminating either a single session or all sessions simultaneously by deleting the corresponding entries from the SESSIONS namespace.

Room management follows Matrix's hierarchical permission model built on power levels, which function similarly to role-based access control but with numeric granularity. Creating a room involves generating a unique room ID, inserting a row into the rooms table, creating initial state events including room creation (m.room.create), power levels (m.room.power_levels), join rules (m.room.join_rules), and optionally a room name, topic, and avatar. The system supports three room presets: public rooms with open joining and public visibility, private rooms requiring invites with private visibility, and trusted private chats optimized for one-on-one or small group conversations with enhanced encryption expectations. Room members can have different power levels ranging from zero (normal user) to one hundred (admin), with specific thresholds required for actions like sending messages, changing room state, kicking users, or banning members. The power levels state event maps specific actions to required power levels and can grant elevated privileges to individual users, creating a flexible permission system that scales from simple private chats to complex community servers with multiple moderator tiers.

The messaging system implements Matrix's event-based data model where everything—messages, state changes, membership updates—is represented as an immutable event with a unique event ID. When a client sends a message, the system validates the user has sufficient power level to send messages in that room, generates a new event ID, constructs the event JSON with the sender, room ID, timestamp, event type (like m.room.message), and content, stores the event in the events table, updates the room's latest event pointer, and notifies all room members through their active sync connections. Events can reference other events through relations, enabling features like threaded conversations where reply events reference their parent, reactions that attach emoji to specific messages, edits that replace previous message content, and annotations that add metadata without modifying the original. The system maintains referential integrity by storing these relationships in a dedicated event_relations table that tracks the parent-child connections and relation types, allowing efficient querying of all replies to a message or all reactions to an event.

The sync mechanism represents one of Matrix's most complex features, providing clients with incremental updates about new messages, state changes, and typing notifications across all their rooms. Traditional sync operates as a long-polling endpoint where clients provide a "since" token representing their last known position in the event stream, and the server responds with all events that occurred since that position. The implementation uses a combination of techniques: for rooms with recent activity, it queries events with timestamps greater than the since token; for newly joined rooms, it sends a complete state snapshot including all current state events and recent message history; and for presence and typing notifications, it checks dedicated tracking tables. The sync response organizes data into several sections: "rooms.join" contains updates for rooms the user has joined, "rooms.invite" holds invitations pending acceptance, "rooms.leave" includes rooms the user recently left, and "presence" and "to_device" provide presence updates and end-to-end encryption messages respectively. The system generates a new sync token representing the current position and returns it to the client, which will use that token for the next sync request, creating a continuous stream of updates.

Sliding sync, defined in Matrix Spec Change 3575, represents a modern alternative to traditional sync that provides more efficient initial synchronization and bandwidth usage. Instead of sending complete data for all rooms on first sync, sliding sync allows clients to request specific "ranges" of rooms sorted by priority—typically by most recent activity—and receive only the most important rooms immediately while lazily loading others as needed. The implementation maintains a sorted list of room priorities per user, handles range requests like "give me rooms 0-19" by querying the twenty most recently active rooms, supports incremental updates using operation codes (INSERT for new rooms, DELETE for removed rooms, INVALIDATE for rooms requiring re-sync), and includes extension mechanisms for end-to-end encryption data, typing notifications, and to-device messages. This approach dramatically reduces the initial sync payload for users in hundreds or thousands of rooms, improving application startup time and reducing bandwidth consumption especially on mobile devices.

End-to-end encryption support follows the Olm and Megolm cryptographic protocols used by Matrix, though the homeserver itself cannot decrypt message content—it merely facilitates key exchange and message delivery. Each device generates identity keys and signs them with the user's account, then uploads these keys to the homeserver which stores them in the DEVICE_KEYS KV namespace. When a user wants to send an encrypted message, their client queries the homeserver for all device keys belonging to room members, establishes Olm sessions for sending one-time encrypted keys, generates a Megolm session key for the room, encrypts the message content with that session key, and sends both the encrypted message and encrypted session keys to the homeserver. The homeserver stores these encrypted blobs without being able to decrypt them and delivers them to recipients through to-device messages and room events. One-time keys provide forward secrecy by allowing each initial key exchange to use a fresh ephemeral key; the system stores uploaded one-time keys in KV and removes them after claiming, ensuring they're truly one-time.

Fallback keys provide a backup mechanism when one-time keys run out, and cross-signing keys enable device verification where users can cryptographically sign their devices and other users' devices to establish trust relationships.

Key backup functionality addresses the challenge of recovering encryption keys across device changes or data loss. Users generate a recovery key and use it to encrypt their Megolm session keys, then upload these encrypted key backups to the homeserver organized by room ID and session ID. The system stores these in D1's key_backup_keys table associated with a version identifier that tracks the backup format and parameters. When a user logs in on a new device, they can retrieve their backed-up keys using the recovery key, decrypt them locally, and regain access to message history without requiring other devices to be online. The implementation includes version management allowing users to create new backup versions, delete old versions, and update version metadata like authentication tags used by some backup algorithms. This strikes a balance between data recovery and security—the homeserver stores encrypted keys but cannot decrypt them without the user's recovery key, maintaining zero-knowledge properties while enabling practical key recovery.

Media handling uses R2 object storage to efficiently store and serve files of arbitrary size. Upload proceeds in two possible modes: the traditional upload where the client sends the file directly in a POST request and receives back a Matrix content URI (mxc://), or the asynchronous upload introduced in MSC3916 where the client first creates a placeholder receiving upload limits and capabilities, then uploads the file data in chunks if needed. The system stores uploaded media in R2 using the content URI's server name and media ID as the key path, preserves the original content type and filename in metadata, and enforces upload size limits (50MB default, configurable). Download endpoints accept the mxc:// URI, look up the corresponding R2 object, and stream it back to the client with appropriate Content-Type headers. Thumbnail generation remains a work-in-progress area; currently the system returns the original image when thumbnails are requested, but plans include either integrating Cloudflare's Image Resizing service (requires Pro plan or higher) or implementing WASM-based image processing to generate common thumbnail sizes on-demand. URL preview functionality implements Open Graph protocol parsing where the server fetches the target URL, extracts metadata like title, description, and image, and caches this preview data for clients to display rich link previews without each client needing to fetch external URLs.

Real-time features like typing notifications, read receipts, and presence require careful coordination to maintain low latency while handling concurrent updates. Typing notifications track which users are currently typing in a room with a timeout mechanism—when a client sends a typing notification, the system records the timestamp in memory or a fast storage layer and automatically clears it after thirty seconds of inactivity. The sync endpoint includes current typing users for each room, allowing clients to display "Alice is typing..." indicators. Read receipts come in two forms: public receipts (m.read) that visible to all room members showing what message each user has read up to, and private receipts (m.read.private) visible only to the user themselves. The system stores the latest read event ID per user per room and uses this to calculate unread message counts and notification states. Presence allows users to broadcast their online status (online, offline, or unavailable) along with a custom status message; the implementation stores presence state and timestamps in D1 and broadcasts changes to users who share rooms with the status-changing user, implementing the privacy model where presence is only visible to users with whom you share a room.

Push notification support follows Matrix's sophisticated push rules system that allows fine-grained control over what generates notifications. Users register pushers—endpoints for receiving push notifications—which can be HTTP pushers that POST to a push gateway, or email pushers for email notifications. Push rules define conditions under which events generate notifications, organized into five priority levels: override rules (highest priority, can silence notifications), content rules (match on message content), room rules (per-room notification settings), sender rules (per-user notification settings), and underride rules (catch-all defaults). The default ruleset includes rules for mentions (@username or @room), direct messages, encrypted messages to the user, and room invites. Each rule specifies conditions (like "sender is X" or "content contains word Y"), actions (notify, set sound, highlight), and an enabled state. When an event arrives, the system evaluates all rules in priority order, executing the actions of the first matching enabled rule and stopping evaluation. This allows users to configure complex notification scenarios like "notify me about all messages in this room, except ones from bot users, but always notify for mentions even from bots."

The rate limiting system protects against abuse while allowing legitimate usage patterns. Implementation uses a sliding window algorithm tracked in KV where each user's recent request timestamps are stored and expired requests are pruned before checking if the limit is exceeded. Different endpoint categories have different limits: authentication endpoints like login and registration have stricter limits (5 requests per minute) to prevent brute force attacks, message sending allows higher throughput (30 messages per minute), and media uploads have both request rate limits and size limits. The rate limit middleware runs early in the request pipeline, extracts the user identifier from the access token or uses IP address for unauthenticated requests, checks the relevant rate limit bucket, and returns HTTP 429 with standard X-RateLimit headers (X-RateLimit-Limit, X-RateLimit-Remaining, X-RateLimit-Reset) when limits are exceeded. The sliding window approach provides more consistent protection than fixed windows since it prevents burst attacks at window boundaries, though it requires storing timestamps for recent requests rather than just a counter.

Administrative functionality includes an HTML dashboard accessible at the root path providing an overview of server statistics, user management, and room exploration. Admin API endpoints allow creating users programmatically with specific power levels and admin flags, viewing and resolving content reports, and sending server notices to users. Content reporting allows users to flag problematic messages or users, storing the report with the reporter's user ID, reported event or user ID, reason code, and free-text explanation in the reports table. Moderators and admins can query reports, view their details, and mark them as resolved. Server notices provide a mechanism for server administrators to send important messages to users through a dedicated notice room that's automatically created and joined, useful for terms of service updates, maintenance notifications, or security alerts. Room upgrades handle the complex process of migrating from an old room version to a new one, creating a new room with updated version, copying over state events and members, creating a tombstone event in the old room pointing to the new room, and optionally migrating room aliases to point to the new room.

Federation represents the most complex unfinished component, though the groundwork exists. Matrix federation uses server-to-server API over HTTPS where servers exchange signed JSON events, enabling users on different homeservers to participate in the same rooms. The current implementation includes placeholder endpoints for server version discovery, server signing key publication, and basic event exchange, but lacks the cryptographic signing and verification required for production federation. Full federation requires implementing Ed25519 signature generation and verification using libraries like @noble/ed25519 or tweetnacl, signing all outgoing events with the server's private key, verifying incoming event signatures using fetched public keys from remote servers, implementing the state resolution v2 algorithm for handling concurrent state changes when servers disagree about room state, and handling the complex join-over-federation flow where a user on one server joins a room hosted on another server. The FederationDurableObject provides a coordination point for managing outbound event queues per destination server, implementing retry logic with exponential backoff for failed deliveries, and batching events into transactions to reduce overhead.

The state resolution v2 algorithm deserves special attention as it's crucial for federation but highly complex. When multiple servers send conflicting state events to a room simultaneously—like two servers both promoting different users to admin status—the servers must reach consensus on which change takes precedence. State resolution v2 performs several steps: calculate the full authorization chain (the sequence of events that authorize each conflicting event), identify the conflicting events and the auth events they depend on, perform reverse topological sort to order events by their dependencies, use mainline ordering based on power level events to break ties, and finally apply lexicographic ordering on event IDs as a last resort. This deterministic algorithm ensures all participating servers arrive at the same state despite seeing events in different orders, maintaining the fundamental property that federation doesn't fork the room into multiple inconsistent versions.

VoIP support leverages Cloudflare's TURN service to provide NAT traversal for WebRTC voice and video calls. The /voip/turnServer endpoint generates time-limited credentials for accessing Cloudflare's TURN servers, which relay RTP packets between clients that cannot establish direct peer connections due to firewalls or NAT. The implementation caches credentials to avoid excessive API calls to Cloudflare's TURN credential endpoint, validates TTLs to ensure credentials remain valid, and returns both STUN and TURN server configurations. STUN servers help clients discover their public IP addresses and port mappings, while TURN servers provide relay functionality as a fallback. The system implements per-user rate limiting on credential generation to prevent abuse, allowing five credential requests per minute per user. While the TURN infrastructure enables basic call setup, full MatrixRTC support for call signaling and multiparty conferences remains unimplemented.

The database schema reflects Matrix's event-sourced architecture where immutable events drive all state changes. The users table stores account information including hashed passwords, display names, avatar URLs, and admin flags. The rooms table tracks room metadata, creation timestamps, and the current room version which determines protocol rules. The events table forms the core of the system, storing every message, state change, and membership update with columns for event ID, room ID, sender, type, state key for state events, content as JSON, depth for topological ordering, and timestamps. The room_members table denormalizes membership state for efficient queries of who's in which room and with what membership status (join, invite, leave, ban, knock). Additional tables handle device information for end-to-end encryption, push rules and pushers, room aliases mapping human-readable names to room IDs, read receipts tracking last-read positions, and account data storing user preferences and settings. The schema uses indexes strategically: covering indexes on events by room and timestamp for efficient room history queries, indexes on room_members for membership checks, and indexes on access tokens for authentication lookups.

Migration management uses Wrangler's D1 execute command with SQL files in the migrations directory. The initial schema.sql creates all base tables, while subsequent migrations add features incrementally: 002_phase1_e2ee.sql adds end-to-end encryption tables for device keys and one-time keys, 003_account_management.sql adds password change history and third-party identifier tables, and 004_reports_and_notices.sql adds content reporting and server notices infrastructure. This incremental approach allows deploying schema changes without disrupting running systems and provides a clear audit trail of database evolution. The D1 platform's transactional guarantees ensure migrations apply atomically—either all changes succeed or all roll back—preventing partial migration states that could corrupt data.

Performance optimization focuses on working within Cloudflare Workers' constraints while maximizing throughput. Workers have CPU time limits that terminate long-running requests, so the implementation prefers simple queries over complex joins, uses KV caching aggressively for frequently-accessed data like user profiles and room state, and offloads expensive operations to Durable Objects which have more relaxed CPU limits. D1 queries use prepared statements with parameter binding to enable query plan caching and prevent SQL injection. The sync endpoint represents a particular challenge since it must query potentially hundreds of rooms; the implementation limits the number of rooms processed per sync request and relies on the client's prioritization to handle large room lists efficiently. R2's global replication provides low-latency media access worldwide without manual CDN configuration. Rate limiting using KV balances protection against abuse with reasonable storage costs by using expiring keys that automatically clean up old request timestamps.

Security considerations permeate the implementation. Passwords undergo bcrypt hashing with appropriate work factors before storage, preventing plain-text exposure even if the database is compromised. Access tokens use cryptographically random generation with sufficient entropy to prevent prediction attacks. SQL injection is prevented through consistent use of parameterized queries rather than string concatenation. Cross-origin resource sharing headers allow web clients to access the API while preventing unauthorized domains from making requests on behalf of users. Input validation sanitizes user-supplied data including usernames, room names, message content, and file uploads, enforcing length limits and type constraints. The end-to-end encryption model ensures the server never has access to message plaintext, providing client-side confidentiality even against server compromise or malicious operators. Content Security Policy headers prevent various injection attacks. The admin flag in the users table controls access to privileged operations, preventing unauthorized users from accessing administrative functions.

Monitoring and debugging leverage Cloudflare's built-in logging with structured output using console.log and console.error. The implementation logs authentication failures, rate limit hits, database errors, and federation events for troubleshooting. Wrangler's tail command provides real-time log streaming during development and production operation. Error responses follow Matrix's standardized format with error codes (errcode field like M_FORBIDDEN, M_NOT_FOUND, M_LIMIT_EXCEEDED) and human-readable error messages, enabling clients to handle errors appropriately and display useful feedback to users. The health check endpoint verifies database connectivity and returns server statistics including uptime and version information.

Testing strategy combines unit tests for utility functions, integration tests for API endpoints using Vitest's Workers integration, and manual testing with Matrix clients like Element X and Element Web. The test suite covers authentication flows, room operations, message sending and retrieval, encryption key management, and edge cases like concurrent room joins or state conflicts. The development workflow uses Wrangler's local development mode which simulates Workers, D1, KV, R2, and Durable Objects locally, allowing rapid iteration without deploying to the cloud. TypeScript provides compile-time type safety, catching many errors before runtime and enabling confident refactoring of the complex codebase.

The live instance at m.easydemo.org demonstrates the system's capabilities, hosting real users and federating with the broader Matrix network (though federation remains limited due to incomplete signing implementation). Test accounts allow experimentation: admin user with administrative privileges and testuser for normal user operations. The instance runs entirely on Cloudflare's free and paid tiers, demonstrating that edge computing platforms can host complex real-time applications traditionally requiring dedicated servers.

Looking forward, the roadmap prioritizes stability and feature completeness: implementing long-polling for sync to reduce client-side polling overhead, completing federation signing and verification to enable true multi-server communication, adding thumbnail generation for better media handling, implementing message search using D1's full-text search capabilities, improving notification count accuracy through better integration with read receipts, and optimizing performance for large room lists. The state resolution algorithm implementation blocks most advanced federation features but represents a significant undertaking due to its complexity. Application service support would enable bridges to other protocols like IRC, Slack, or Discord, dramatically expanding interoperability. Advanced caching strategies using KV could further reduce database load and improve response times.

This implementation demonstrates that serverless and edge computing platforms can support protocols designed for traditional server architectures, though not without careful design consideration. The stateless nature of Workers requires offloading stateful operations to Durable Objects. The CPU time limits necessitate query optimization and careful algorithm selection. The distributed nature of edge computing requires thinking about data locality and replication. Yet the benefits are substantial: global distribution provides low latency worldwide, automatic scaling handles traffic spikes without provisioning, pay-per-use pricing makes small deployments economical, and Cloudflare's security features provide DDoS protection and WAF capabilities without additional infrastructure. Tuwunel Workers proves that with thoughtful architecture, even complex federated communication protocols can run effectively on edge computing platforms, potentially offering better performance and lower operational costs than traditional deployments.

Document Version: 1.0 (Narrative Format)
Last Updated: January 14, 2026
Project Status: Active Development
Live Instance: https://m.easydemo.org