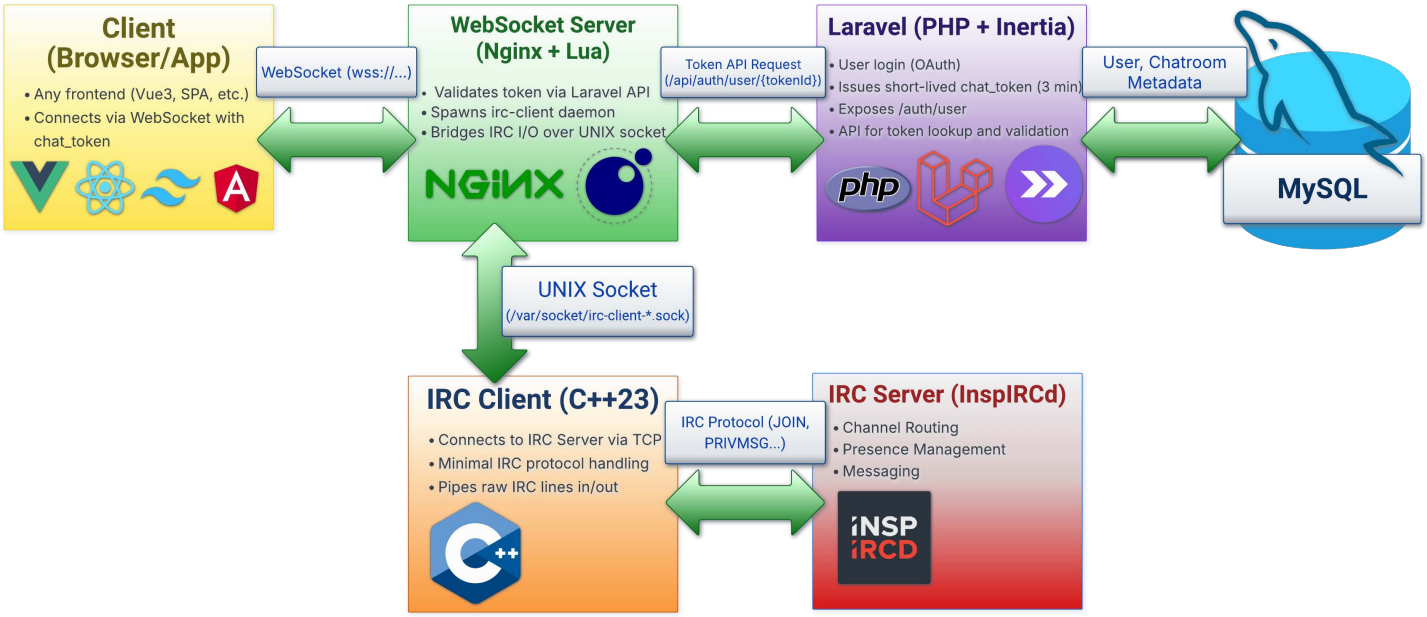
eIRC: Technical Architecture Overview

eIRC is a modern, vertically-scalable enterprise messaging architecture built on the IRC protocol. Designed for organizations that require ephemeral, real-time communication without the heavy operational overhead of pub/sub systems like Apache Kafka, eIRC delivers high-throughput, low-latency chat experiences while minimizing memory and CPU usage per user.



Full Image: (https://i.imgur.com/d5CblBK.png)

The system is composed of three core architectural tiers:

1. Frontend Client (any browser-based or native application)
2. Session Bridge Layer (WebSocket server running in OpenResty, per-user IRC client process)
3. Infrastructure Services (IRC server, Laravel-based backend, persistent database)

This whitepaper outlines the full technical architecture, component responsibilities, communication paths, deployment model, and scalability considerations.

Architecture Diagram

The architecture is represented using a C4 Container Diagram. The core containers include:

- Browser Client
- WebSocket Server (OpenResty + Lua)
- IRC Client (C++)

- IRC Server (InspIRCd)
- Laravel Backend (PHP + Inertia)
- PostgreSQL or MySQL database

---

1. Frontend Client

Description:

The frontend is a JavaScript application built with Vue 3 and Inertia.js. However, eIRC is frontend-agnostic: any application capable of opening a WebSocket and handling IRC-style messages can integrate with the backend.

Responsibilities:

- Authenticate user via OAuth2 (Laravel Passport)
- Request short-lived chat_token from Laravel (/chat route)
- Open WebSocket connection to OpenResty using chat_token
- Display incoming IRC messages
- Send user input as raw IRC protocol lines (e.g., PRIVMSG, JOIN)

---

2. Session Bridge Layer

WebSocket Server (OpenResty + Lua)

Description:

Built inside Nginx workers using the lua-resty-websocket and ngx.pipe libraries, this layer bridges the frontend and backend without external WebSocket daemons or Node.js servers.

Responsibilities:

- Accept WebSocket upgrade requests
- Validate chat_token via internal API call to Laravel (/api/auth/user/{tokenId})
- Spawn a per-user irc-client process using ngx.pipe
- Connect to the IRC client via a UNIX socket
- Stream IRC messages from the client to the browser

- Forward browser input to the IRC client
- Handle lifecycle cleanup (WebSocket close, IRC process shutdown)

IRC Client (C++23)

Description:

A small, memory-efficient IRC protocol proxy. Each user connection results in one instance of this process. It connects to the IRC server and communicates with the WebSocket server over a UNIX domain socket.

Responsibilities:

- Authenticate and connect to InspIRCd
- Join channels as directed
- Parse IRC messages and return them via UNIX socket
- Accept frontend commands via socket and translate to IRC protocol
- Shut down cleanly on quit or socket disconnect

Resource Usage:

- 10 MB resident memory per process
- Minimal CPU load during idle; spiky under large IRC traffic bursts
- Native compiled with ASIO for networking and Ncurses/Unix socket for I/O abstraction

---

3. Infrastructure Services

IRC Server (InspIRCd)

Description:

A standard IRC daemon which handles routing, presence, and channel state. No modification required. It is treated as a pure message bus.

Responsibilities:

- Handle all IRC protocol delivery
- Manage channels, users, joins, parts, and messages
- Coordinate presence and mode tracking

- Optionally clustered (hub-spoke or spanning tree)

Laravel Backend

Description:

A Laravel 12 application responsible for persistent user management, chatroom metadata, and issuing short-lived tokens.

Responsibilities:

- User authentication via Laravel Passport
- Chatroom metadata API
- OAuth-based login and token issuance
- Serve Inertia-powered frontend for chat UI
- Issue and invalidate chat_token for WebSocket auth

Database (PostgreSQL or MySQL)

Responsibilities:

- Store user profiles, nicknames, preferences
- Track persistent chatroom metadata
- Store issued access tokens for temporary chat access
- Optionally log IRC messages (for analytics or compliance)

Token-Based Authentication Flow

1. User logs in via Laravel UI
2. Laravel issues a short-lived chat_token (valid for 3 minutes)
3. Browser sends WebSocket connection request with ?chat_token=xyz
4. WebSocket server validates token via internal HTTP API to Laravel
5. If valid, binds token to Nginx request_id in ngx.shared.DICT
6. Spawns IRC client process for that session

Tokens are:

- Bound to one connection only
- Deleted after use or timeout
- Prevent session reuse across tabs/devices

Message Flow Diagram

1. Browser ⟶ WebSocket: wss://server/chat?chat_token=abc
2. WebSocket ⟶ Laravel API: /api/auth/user/{tokenId}
3. WebSocket ⟶ Spawn IRC client via ngx.pipe
4. WebSocket ⟶ Connect to irc-client.sock
5. IRC Client ⟶ InspIRCd: CONNECT, NICK, USER, etc.
6. IRC ⟶ IRC Client ⟶ WebSocket ⟶ Browser

Scalability Considerations

Vertical Scaling

- WebSocket server runs inside Nginx workers (~20 MB each)
- IRC clients: 10 MB each × N users = predictable memory curve
- Suitable for bare-metal or virtual Linux hosts with high thread count

Horizontal Scaling

- Stateless WebSocket workers can be scaled across machines
- Token binding is enforced per instance via shared memory (per Nginx worker)
- Laravel backend and database can be clustered
- IRC server clustering is supported via InspIRCd's native configuration

---

Deployment Model

Recommended Setup:

- 1 physical or VM host
- Nginx + OpenResty
- PHP-FPM for Laravel
- Redis (optional, for central cache)
- InspIRCd daemon

Processes:

- 1 Nginx master + N workers
- 1 PHP-FPM pool
- 1 irc-client process per active user

Optional Extensions

- Message Logging: Use a logger bot to stream messages to Elasticsearch or Redis Streams
- IRC History Bridge: Implement Redis-backed buffer for message replay
- UI Clients: Native mobile apps, Electron clients, embedded widgets
- Audit Trails: Laravel logs all chat_token issuances and user metadata updates

---

Advantages Over Kafka-Based Systems

Feature	eIRC (IRC-Based)	Kafka-Based System
Memory per connection	~10MB	High (JVM overhead)
Delivery latency	<10ms	~100ms+
Storage requirements	None (ephemeral)	Always-on logging
Protocol complexity	Simple text-based IRC	Avro/Protobuf, partitions
Fan-out/multiplexing	Via InspIRCd clustering	Native pub/sub
Replay	Not built-in	Built-in
Cost per message	Extremely low	High (CPU + I/O)

Summary

eIRC offers a highly efficient alternative to heavyweight enterprise messaging systems. By leveraging IRC's lightweight protocol, combining it with modern WebSocket infrastructure, and introducing per-user process isolation, it achieves:

- Portability
- Ephemeral real-time communication
- Low infrastructure cost
- Easy integration into existing frontends

This system is best suited for organizations seeking a chat layer that is:

- Ephemeral by design
- Not reliant on persistent log streams
- Lightweight enough for embedded use
- Secure through token-based, OAuth-integrated flows

Minimum System Requirements

- Linux host with:
    - 4+ CPU cores
    - 8+ GB RAM (for ~500 simultaneous users)
    - SSD storage for logs and tokens
- Installed services:
    - OpenResty (with lua-resty modules)
    - InspIRCd
    - PHP 8.3 + Laravel 12
    - Redis (optional)
    - PostgreSQL or MySQL

Contact

For technical consultation or implementation support, please contact:

Jesse Greathouse
Email: jesse.greathouse@gmail.com
GitHub: https://github.com/jesse-greathouse/eIRC