TECHNICAL PLATFORM REPORT

LOLdle

A League of Legends Champion-Guessing Game

Full-Stack Web Application • Wordle-Style Attribute Comparison • Containerized Deployment

Edgardo Paz-Romero
Computer Science • Temple University
Full-Stack Engineering • API Design • DevOps
loldle.fly.dev

7	10	10 Days	Live
Compared Attributes	Guess Limit	Challenge TTL	loldle.fly.dev

Abstract

LOLdle is a full-stack web application that brings the Wordle format to the League of Legends champion roster. A creator selects a champion and generates a shareable link; recipients visit the link and attempt to guess the champion through Wordle-style attribute feedback across seven dimensions — exact matches for scalar fields, set-intersection logic for array fields, and proximity signals for release year. The system enforces all game-state rules server-side: duplicate and invalid guesses are rejected without consuming attempts, challenge expiration is TTL-based, and the defeat condition spans two distinct API response shapes to preserve the final comparison row. Player state lives in localStorage, making each challenge stateless on the backend and shareable across multiple players. The application is fully containerized and deployed on Fly.io with a persistent SQLite volume, Helmet.js security hardening, and a layered Express architecture separating route handling from service and data logic.

Table of Contents

1. Introduction & Problem Statement
2. System Architecture Overview
3. Game-State Engine
4. Attribute Comparison Logic
5. API Contract Design
6. Data Model & SQLite Infrastructure
7. Frontend Architecture
8. Security & Production Hardening
9. Deployment & Infrastructure
10. Notable Issues & Resolutions
11. Lessons Learned
12. Technology Stack Summary

1. Introduction & Problem Statement

Wordle demonstrated that a simple, shareable daily puzzle with clear feedback mechanics could generate massive engagement. LOLdle applies that format to League of Legends — a game with a roster of over 160 champions, each with a rich set of comparable attributes — creating a guessing game that is familiar in structure but requires domain knowledge to play well.

The core challenge in building LOLdle was not the concept but the engineering constraints it implied: game-state rules needed to be enforced server-side to prevent cheating, the shareable-link model required separating challenge identity from player state, the attribute comparison engine needed to handle heterogeneous data types consistently, and the deployment target needed to support a persistent file-based database inside an ephemeral container runtime.

This report details the architecture, design decisions, and production issues encountered across approximately one month of development.

---

2. System Architecture Overview

LOLdle is organized around a layered Express backend, a two-page vanilla frontend, and a SQLite database managed via better-sqlite3. The architecture prioritizes simplicity and correctness over scale — appropriate for a personal project — while still applying production engineering practices: separated service layers, stable API contracts, graceful shutdown, and security hardening.

LOLDLE — SYSTEM ARCHITECTURE

CLIENT (Vanilla HTML / CSS / JS)			
Home (/) Challenge Creator	**Play (/play.html?id=...)** Guess Interface • localStorage owns player run		
EXPRESS SERVER (Node.js)			
Route Layer HTTP, validation, response shaping	**Service Layer** Game logic, comparison, expiration	**Middleware** Helmet, rate-limit, error handler	**Data Layer** better-sqlite3, synchronous queries
SQLite (persistent Fly.io volume) challenges table • guesses table		**Riot Data Dragon (champion data)** name mapping • square icons • attributes	

Figure 1 — System architecture: client, server layers, persistence, and external data.

The critical path for a player guess — validation, game-state check, attribute comparison, response shaping — executes synchronously. Challenge expiration cleanup (lazy deletion of expired rows) fires as a side effect of challenge creation, keeping the architecture free of background job infrastructure.

---

3. Game-State Engine

All game-state rules are enforced server-side. The frontend reflects state but does not own it. On every guess submission, the backend evaluates the following checks in order:

3.1 Validation Rules

- Invalid champion name — unrecognized names are rejected with a 400 error. The attempt counter is not decremented.
- Duplicate guess — re-submitting a previously guessed champion is rejected. No attempt consumed.
- Expired challenge — challenges past their 10-day TTL respond as 404. No guess processing occurs.
- Solved challenge — once the correct champion is guessed, subsequent submissions are rejected immediately.
- Guess limit overflow — attempts after the 10-attempt limit return 400 with code: 'guess_limit_reached' and the revealed champion.

3.2 Guess Limit & Defeat States

The defeat condition has two distinct shapes. The 10th wrong guess returns HTTP 200 with status: 'ok', remainingGuesses: 0, and the revealed champion — preserving the final attribute comparison row in the UI. A subsequent overflow attempt returns HTTP 400 with code: 'guess_limit_reached'. This split was deliberate: collapsing both into a 400 would have prevented the last comparison row from rendering, degrading the end-of-game experience.

---

4. Attribute Comparison Logic

The core game mechanic compares a guessed champion against the answer across seven attributes. Each attribute uses a different comparison strategy based on its data type:

gender	Exact match — correct or incorrect
position	Set intersection — partial if any roles overlap; correct if sets are equal
species	Set intersection — partial if any species tags overlap; correct if sets are equal
resource	Exact match — mana / energy / none / other
rangeType	Exact match — melee or ranged
region	Set intersection — partial if any regions overlap; correct if sets are equal
releaseYear	Proximity — directional arrow (earlier / later) + exact match

Array-typed fields (position, species, region) use set-intersection: a cell is marked partial if the guessed value shares any elements with the answer, and fully correct only if the sets are identical. The release year arrow gives the player a directional signal — reducing the search space on each guess — without revealing the answer.

The share output encodes one emoji box per attribute, producing exactly 7 boxes. An early implementation risked producing 8 by including a synthetic name field; catching this required auditing the comparison pipeline against the intended share format.

---

5. API Contract Design

5.1 Response Shape Stability

The frontend was refactored to branch on stable backend code fields rather than matching human-readable error message strings. This ensures behavior stays predictable across API changes and makes integration tests assertable on contract values rather than brittle text. Tests assert on code, remainingGuesses, and solved fields — not on message text.

5.2 Secure Champion Reveal

The answer is never sent to the client on a normal guess response. The backend returns attribute comparison results only. The champion is revealed in two controlled cases: on a correct guess (solved: true), and on terminal loss — both the 10th wrong guess (200, remainingGuesses: 0) and the overflow guard (400, guess_limit_reached). Once the backend became stateless for guesses (player state moved to localStorage), the reveal logic had to be redesigned: the server evaluates each guess independently and includes the answer only when the guess itself produces a terminal state.

5.3 Frontend Resilience

The frontend originally assumed all API responses contained valid JSON. Safe parsing with fallback error handling was added so that proxy errors, malformed responses, or server restarts during a request do not crash the play page mid-session.

---

6. Data Model & SQLite Infrastructure

6.1 Schema Design

Challenges and guesses are stored in separate tables. This separation — rather than overloading a single table — makes guess counting, solved-state detection, and per-challenge history retrieval straightforward without complex queries. The challenges table stores the champion answer, share token, creation timestamp, and TTL expiration. The guesses table records each guess attempt associated with a challenge.

6.2 Expiration Strategy

Challenges use a 10-day TTL. Expired challenges respond as 404 rather than introducing a new API error shape — this keeps the client error-handling surface small. Physical row deletion is handled lazily: expired rows are deleted as a side effect of challenge creation, avoiding the operational overhead of a background cleanup job.

6.3 Persistent Volume on Fly.io

Fly.io containers are ephemeral by default. Without a persistent volume, the SQLite file is wiped on every deployment or container restart. A Fly.io volume is mounted at the database file path, declared in fly.toml and attached automatically on each deploy. The DB path is configurable via environment variable, so the same Docker image runs locally against a local file and in production against the mounted volume. Verifying that the deployed application was actually writing to the volume — rather than to a container-local path — required checking usage against the deployed database directly.

---

7. Frontend Architecture

7.1 Per-Player State in localStorage

An early architectural decision stored guess history server-side, associated with a player session. This broke the intended sharing model: one link sent to a group of friends would have all players sharing the same guess history. The fix moved all player-owned state — guesses, solved status, and end-game stats — to localStorage. The backend became stateless with respect to individual player sessions: each guess is evaluated independently, and the server never stores who made it.

7.2 Stats Deduplication

End-of-game stats are stored in localStorage. Without deduplication, refreshing the page or re-entering a completed challenge would double-count wins. A completedChallengeIds set was added to prevent duplicate recording. The set is capped in size to prevent unbounded localStorage growth.

7.3 UI Details

Several frontend details required more precision than expected:

- Autocomplete behavior — champion name suggestions needed correct filtering and keyboard navigation without interfering with the guess submission flow.
- Modal CSS scoping — the end-of-game stats modal introduced selector leakage that caused floating UI artifacts after wrong guesses. Fix: all modal styles scoped under #statsModal only.
- Share/copy fallback — the share button uses the Clipboard API with a fallback for browsers that do not support it.
- Release year arrow — directional feedback (earlier / later) rendered inline in the result cell alongside the year value.
- Cache invalidation — after cache headers were added to static assets, stale play.js caused frontend changes to appear broken in production. Cache behavior became part of the deployment debugging workflow.

---

8. Security & Production Hardening

8.1 Helmet.js & CSP

Helmet.js is applied as Express middleware to set HTTP security headers. Adding Helmet introduced a Content Security Policy breakage in production: Riot Data Dragon champion images and external fonts were blocked by the default policy. The fix was a targeted CSP allowlist for the required external origins rather than disabling Helmet entirely.

8.2 Rate Limiting & trust proxy

express-rate-limit was added to limit guess submission frequency. Setting trust proxy: true to enable correct IP resolution triggered a permissive proxy warning that weakened the IP-based limiting. The fix was a bounded trust model — specifying the exact number of trusted proxy hops — which restored correct client IP resolution without the warning.

8.3 Answer Security

The challenge answer is stored server-side and associated with an opaque share token. The token is the only value in the shareable URL. Recipients never receive the answer in any normal API response — it is included only on terminal game states (correct guess or final loss), at which point the game is over.

---

9. Deployment & Infrastructure

9.1 Docker

The application is fully containerized. The Dockerfile defines the Node.js runtime, installs production dependencies, and sets the entrypoint. Containerization ensures environment parity between local development and production and makes the deployment process reproducible.

9.2 Fly.io

Deployment is triggered via fly deploy, which builds and ships the container image with zero-downtime rolling updates. The persistent SQLite volume is declared in fly.toml and mounted automatically. The health route (/health) is used by Fly.io for uptime checks and rolling deploy readiness signals.

9.3 Graceful Shutdown

The server handles SIGTERM and SIGINT to close the SQLite connection before the process exits. Fly.io sends SIGTERM during rolling deployments — without a handler, in-flight writes could corrupt the database file on the persistent volume.

---

10. Notable Issues & Resolutions

Issue	Root Cause	Resolution
better-sqlite3 native module mismatch	Node version change caused the native module to segfault or fail to load at startup	npm rebuild better-sqlite3 --build-from-source against the correct Node ABI; pin Node version in Dockerfile
Frontend / backend defeat reveal drift	Frontend only handled 400 overflow-loss; 10th wrong guess (200, remainingGuesses: 0) did not reveal champion	Both terminal-loss shapes now include champion in response; frontend handles both paths explicitly
Modal CSS leakage	End-of-game modal selectors leaked into the main page, causing floating UI artifacts after wrong guesses	Scoped all modal styles under #statsModal; no global selectors in modal stylesheet
Stats double-counting on refresh	Re-entering a completed challenge URL re-triggered win recording in localStorage	Added completedChallengeIds set with a size cap to deduplicate completions across sessions
Data Dragon name mapping	Riot champion icon filenames use internal IDs, not display names — special characters broke asset lookups	Manual mapping table for K'Sante, Bel'Veth, Wukong, Aurelion Sol, and other edge cases

Helmet CSP blocking assets	Default Helmet CSP blocked Riot CDN images and external fonts in production	Added targeted CSP allowlist for required external origins; Helmet otherwise retained
trust proxy weakening rate-limit	trust proxy: true triggered permissive proxy warning and weakened IP-based rate limiting	Switched to bounded trust model (hop count) to restore correct client IP resolution
Share output shape (8 vs 7 boxes)	Early comparison pipeline included a synthetic name field, producing 8 share boxes instead of 7	Removed name from comparison output; share format now matches the 7 compared attributes exactly

---

11. Lessons Learned

11.1 Separate Player State from Challenge State

The most significant architectural decision was moving player-owned state (guess history, solved status, stats) from the server to localStorage. The original server-side model conflated challenge identity with player session, breaking the one-link-for-multiple-players use case. Stateless backend evaluation makes sharing natural and eliminates a class of session management bugs.

11.2 Terminal States Require Explicit API Design

The two defeat response shapes (200 final miss vs. 400 overflow) were not incidental — they reflected a real distinction with different UI consequences. Terminal states in a game API deserve the same design attention as the happy path. Leaving them implicit will produce frontend inconsistencies that are hard to trace.

11.3 Native Modules Are a Deployment Risk

better-sqlite3 compiles against a specific Node ABI. Any Node version change — even a minor update — can silently break it. The recovery path (rebuild from source) is well-documented, but the failure is non-obvious at runtime. Projects using native modules should pin Node versions tightly in the Dockerfile.

11.4 Production Infrastructure Has Different Failure Modes

Several issues only appeared in the deployed environment: CSP blocking assets, rate-limit misconfiguration, cache staling after deploys, and SQLite path resolution against the mounted volume. Debugging production required understanding the gap between local and deployed environments — not just reading error messages at face value.

11.5 Document Design Decisions at the Fork

This project was built over a month of iterative development without contemporaneous documentation. Reconstructing the rationale behind the TTL model, the trust proxy configuration, and the per-player localStorage model was significantly harder after the fact. Brief decision notes at each significant fork would have made this report easier to write and the codebase easier to revisit.

---

12. Technology Stack Summary

Frontend	Vanilla HTML / CSS / JavaScript — no framework
Backend	Node.js, Express.js
Database	SQLite via better-sqlite3 (synchronous queries)
Containerization	Docker
Hosting	Fly.io with persistent volume
Security	Helmet.js (CSP + HTTP headers), express-rate-limit
Champion Data	Riot Games Data Dragon API with manual name mapping
Comparison Engine	7-attribute: exact / set-intersection / proximity
Player State	localStorage (guess history, stats, solved state)
Deployment	fly deploy — rolling updates, zero downtime
Health / Shutdown	/health route, SIGTERM graceful close
Testing	Assertions on code fields and counters, not message text