DrawChain

Technical Design Document

Version 1.0

System Architecture

High-Level Architecture

DrawChain follows a client-server architecture with real-time WebSocket communication. The system is designed for horizontal scalability and uses a microservices approach for independent component scaling.

Core Components

- Frontend (React SPA): User interface and drawing canvas
- WebSocket Server (Node.js): Real-time game state synchronization
- API Server (Node.js/Express): REST endpoints for room management
- Redis: In-memory cache for active game state and session management
- PostgreSQL: Persistent storage for completed games and analytics
- S3/Cloud Storage: Drawing image storage

Frontend Architecture

Technology Stack

- React 18.x with TypeScript
- Vite for build tooling and dev server
- Zustand for state management
- Socket.io-client for WebSocket communication
- Tailwind CSS for styling
- HTML5 Canvas API for drawing

Component Structure

App.tsx: Root component with routing

HomePage.tsx: Landing page with room creation/joining

LobbyPage.tsx: Pre-game lobby with player list and settings

GamePage.tsx: Main game interface container

DrawingCanvas.tsx: Canvas component with drawing tools

DescriptionInput.tsx: Text input for descriptions

ResultsPage.tsx: Chain reveal and sharing interface

Timer.tsx: Countdown timer component

State Management

Zustand stores are organized by domain with the following key stores: gameStore (current game state, players, turn info), socketStore (WebSocket connection management), canvasStore (drawing history, undo/redo stack), and uiStore (modals, notifications, loading states).

Drawing Canvas Implementation

The drawing canvas uses HTML5 Canvas API with optimizations for smooth rendering and minimal lag. Key features include:

- Mouse and touch event handling with unified interface
- Line smoothing using quadratic curves for natural drawing
- Brush pressure simulation based on drawing speed
- Undo/redo using snapshot-based history
- Canvas export to PNG using toDataURL for storage

Backend Architecture

Technology Stack

- Node.js 20.x with TypeScript
- Express.js for REST API
- Socket.io for WebSocket server
- Redis for session storage and caching
- PostgreSQL with Prisma ORM
- AWS S3 for image storage

API Endpoints

Method	Endpoint	Description
POST	/api/rooms	Create new game room
GET	/api/rooms/:id	Get room details
POST	/api/rooms/:id/join	Join existing room
POST	/api/upload	Upload drawing to S3
GET	/api/prompts/random	Get random prompt

WebSocket Events

Event	Description

player:joined	Broadcast when player joins room
player:left	Broadcast when player leaves room
game:start	Host initiates game start
turn:next	Advance to next player turn
submission:drawing	Player submits drawing
submission:description	Player submits description
game:complete	All turns complete, show results

Database Schema

PostgreSQL Tables

rooms

Stores completed game room data for analytics and potential future replay features.

Columns: id (UUID), created_at (timestamp), completed_at (timestamp), player_count (integer), initial_prompt (text), room_code (string)

chain_entries

Individual entries in the telephone chain (drawings or descriptions).

Columns: id (UUID), room_id (UUID FK), sequence_number (integer), entry_type (enum: drawing/description), content (text for descriptions, S3 URL for drawings), player_name (string), created_at (timestamp)

prompts

Library of game prompts with metadata.

Columns: id (UUID), text (string), category (string), difficulty (enum: easy/medium/hard), usage_count (integer)

Redis Data Structures

Active game state: Hash storing current game state (room:{roomId}:state)

Player sessions: Hash for player connection info (room:{roomId}:players)

Chain data: List storing entries in order (room:{roomId}:chain)

TTL: All Redis keys expire after 24 hours

Security & Privacy

Authentication & Authorization

- No user accounts required for MVP
- Room access controlled by unique room codes
- Socket connections validated against room membership

Input Validation & Sanitization

- Player names: max 20 characters, alphanumeric only
- Descriptions: max 200 characters, filtered for profanity
- Drawing size: max 2MB per image
- Rate limiting: 5 submissions per minute per player

Content Moderation

- Text descriptions run through profanity filter
- Report functionality for inappropriate content
- Manual review queue for reported content

Performance Optimization

Frontend Optimizations

- Code splitting by route for faster initial load
- Canvas rendering on requestAnimationFrame
- Image compression before upload (quality 0.8, max 1024x1024)
- Debounced WebSocket emissions

Backend Optimizations

- Redis for hot data, PostgreSQL for cold storage
- Connection pooling for database connections
- CDN for static assets and uploaded images
- Horizontal scaling with load balancer

Deployment Architecture

Infrastructure

- Frontend: Vercel or Netlify for automatic deployments
- Backend: AWS EC2 with auto-scaling groups
- Database: AWS RDS for PostgreSQL
- Cache: AWS ElastiCache for Redis
- Storage: AWS S3 with CloudFront CDN
- Load Balancer: AWS ALB with sticky sessions

CI/CD Pipeline

- GitHub Actions for automated testing and deployment
- Staging environment for pre-production testing
- Blue-green deployments for zero downtime
- Automated rollback on deployment failures

Monitoring & Logging

Metrics

- Active WebSocket connections
- Average game completion time
- API response times
- Error rates and types
- S3 upload success rate

Tools

- CloudWatch for infrastructure metrics
- Sentry for error tracking
- LogRocket for session replay on bugs

Testing Strategy

Frontend Testing

- Unit tests: Vitest for component logic
- Integration tests: React Testing Library
- E2E tests: Playwright for critical user flows

Backend Testing

- Unit tests: Jest for business logic
- Integration tests: Supertest for API endpoints
- Socket tests: Socket.io client for WebSocket flows

Load Testing

- Artillery.io for WebSocket load testing
- Target: 100 concurrent rooms (600 players)
- Test scenarios: room creation, gameplay, simultaneous completions

Future Technical Considerations

- WebRTC for peer-to-peer drawing synchronization
- Mobile app using React Native with shared business logic
- Machine learning for content moderation

- GraphQL API for flexible data fetching
- Blockchain for provable ownership of viral chains (NFTs)

End of Technical Design Document