# Phase 1: Market & Technical Landscape Analysis - Strategic Context

Research Goal: Understand how collaborative whiteboards reached product-market fit, what infrastructure actually costs at scale, and which technical decisions separate successful products from failed ones.

## Launch Scale & Growth Trajectories: Why Modest Starts Matter

### The Pattern: Every Major Player Launched Small

None of the category leaders launched with massive user bases. This contradicts the common startup assumption that you need immediate viral growth to succeed.

Miro (founded 2011): Took **7 years** to reach 2 million users by their 2018 Series A. They had just 6,000 paying customers at that point. This is a $17.5B company that spent nearly a decade in slow growth.

Figma (launched 2016): Didn't cross 4 million users until 2022—**6 years post-launch**. Now valued at $68B in their 2025 IPO. Their growth was linear for years before going exponential.

Excalidraw (2020): Started as a weekend project. Grew from 60K to 170K monthly active users in year two. Reached 850K MAU entirely bootstrapped, proving you don't need VC funding to reach scale.

tldraw (2021): Hit 500K MAU on just $14.1M in total funding. Built as an SDK first, product second—monetizing through developer licensing rather than end users.

### Why This Pattern Exists

Product complexity creates high friction. Collaborative whiteboards require behavior change—teams must abandon physical whiteboards, PowerPoint, or paper. This isn't a "10x better email client" where adoption is intuitive. Users need onboarding, training, template libraries, and integration with existing workflows.

Network effects take time to materialize. A whiteboard with one user is worthless. Value compounds as teams adopt it together. This creates slow initial growth followed by hockey-stick acceleration once critical mass hits within organizations.

The market wasn't ready until COVID. Pre-2020, remote work was niche. Collaborative tools were "nice to have." The pandemic forced distributed teams, creating urgent demand overnight.

### The COVID Inflection Point Changed Everything

Miro's growth: Added **1 million MAU in Q2 2020 alone**. Grew from 5M to 30M users (April 2020 → January 2022). That's 500% growth in 21 months after 9 years of linear growth.

MURAL: Tripled ARR two years in a row during the pandemic.

Why this matters for your product: The market is now mature and competitive. The "COVID bonus" is gone. New entrants can't expect pandemic-level growth rates. You're competing against established products with 10+ years of accumulated features, templates, and integrations.

### Current Scale Comparison

Product	Users	Revenue	Funding	Valuation	Strategy
Miro	100M+	~$500M ARR	$476M	$17.5B	Freemium → Enterprise
Figma	13M MAU	$749M (2024)	$749M	$68B	Prosumer → Enterprise
MURAL	Enterprise	~$189M ARR	$199M	$2B	Enterprise-first
Excalidraw	850K MAU	Bootstrapped	$0	N/A	Open-source + Plus tier
tldraw	500K MAU	SDK licensing	$14.1M	N/A	Developer tool

Strategic implication: Three viable paths exist—freemium with enterprise upsell (Miro), developer-first (tldraw), or pure open-source with paid features (Excalidraw). The enterprise-first approach (MURAL) requires significant upfront capital.

## Traffic Patterns: Why Architecture Must Handle Spikes

### The Reality: Whiteboards Have Spiky, Not Steady-State Traffic

Usage clusters during business hours (9AM-6PM per timezone) and **surges 10-50x during planned workshops**. A board might have zero active users for weeks, then 200 concurrent users for a 2-hour PI planning session.

Why this pattern exists:
- Teams schedule brainstorming sessions, retrospectives, and workshops
- Spontaneous collaboration is rare—most usage is calendar-driven
- Peak load is highly predictable (Monday mornings, quarterly planning)

Figma's data: 80%+ of weekly active users are outside the US. Traffic literally follows the sun—US East Coast peak → EU peak → Asia peak → US West Coast peak in a 24-hour cycle.

### Architectural Implications

Traditional steady-state infrastructure is wasteful. Provisioning for peak load means 90% idle capacity most hours. Serverless architectures (Cloudflare Workers, AWS Lambda) are optimal because you pay per-request, not per-server-hour.

Session-based burst capacity is the design constraint. The system must scale from 0 to 200 concurrent users in under 60 seconds. Autoscaling based on CPU/memory is too slow—by the time new servers provision, the workshop has already started and users are experiencing lag.

Connection affinity matters. All users editing the same document must connect to the same server instance (or Durable Object). Load balancers need sticky sessions based on document ID, not round-robin distribution.

## Pricing Convergence: Why Everyone Charges $8-16 Per Seat

### The Market Has Settled on Identical Pricing Tiers

Product	Free	Starter/Team	Business	Enterprise
Miro	3 boards	$8/user/mo	$16/user/mo	~$20-28/user
Figma	3 files	$16/user/mo	—	$55-90/user
MURAL	3 murals	$10/user/mo	$18/user/mo	Custom
Excalidraw+	Unlimited	$6/user/mo	—	—

Why this convergence happened:

$8-10/user is the minimum viable price. Below this, transaction fees (Stripe's 2.9% + $0.30) and customer acquisition costs exceed revenue. A $5/month subscription loses $0.30 to Stripe (6%) plus $2-5 in marketing cost to acquire the customer.

$16-20/user is the "business tier" equilibrium. Teams will pay this much for a tool they use daily. Above $25/month, you're competing with enterprise software budgets, requiring sales teams and procurement processes.

Free tiers are limited to 3 boards/files because this allows personal use (sandbox, portfolio) while forcing teams to upgrade. Teams need 10-50+ boards for different projects. The "3 board limit" is precise—enough to try the product, not enough to run a team.

### Why You Can't Compete on Price

Undercutting ($3-5/user) signals low quality. Users assume "cheap" means "limited features" or "will shut down soon." Collaborative tools have high switching costs—teams won't adopt a product that might disappear.

Premium pricing ($25+/user) requires premium features. Miro charges $16/user for business tier. To charge $25, you'd need features Miro lacks. As a new entrant, you don't have feature parity yet.

The $8-12/user range is optimal for MVP. Matches market expectations, covers infrastructure costs (see next section), and leaves margin for customer acquisition.

## Infrastructure Economics: Why Collaborative Apps Stay Cheap

### The Surprising Truth: Real-Time Costs Less Than You Think

A collaborative whiteboard serving **100 concurrent users** costs **$100-260/month** across compute, database, storage, CDN, Redis, and bandwidth. This is shockingly low compared to video streaming or ML inference workloads.

At 1,000 concurrent users: $425-1,150/month
At 10,000 concurrent users: $2,500-6,800/month

### Why Real-Time Infrastructure Is Cheap

WebSocket connections are mostly idle. A connection consumes 50-200KB of memory when idle. Active drawing generates 30-60 messages/second, but users spend 90% of their time reading (looking at the board) not writing (drawing).

Message payloads are tiny. A cursor position update is 20 bytes: `{userId, x, y}`. An object move is 50 bytes: `{objectId, x, y, version}`. Even at 60 messages/second, bandwidth is negligible (~3KB/second per user).

Compute is bursty but predictable. CPUs sit idle when no one is drawing. They spike to 80% when 50 users draw simultaneously. But collaborative sessions have natural turn-taking—rarely do all users draw at once.

Storage scales linearly. A board with 500 objects (sticky notes, shapes, text) is ~100KB. Even with version history (10 snapshots), storage is 1MB per board. At $0.023/GB-month (S3 pricing), you can store 43,000 boards per dollar per month.

### Cost Breakdown at 1,000 Users

Component	Monthly Cost	Why This Cost
Cloudflare Workers	$50-100	WebSocket routing + static assets
Durable Objects	$40-80	Per-room state management
Supabase (Postgres)	$25	Metadata, users, permissions
Upstash Redis	$15-30	Presence, pub/sub, rate limiting
R2 Storage	$5-10	Board snapshots, uploaded images
Bandwidth	$10-20	CDN + WebSocket messages
Total	**$425-1,150**	

Break-even calculation: $425/month ÷ 1,000 users = $0.43 per user per month in infrastructure costs. With $10/month pricing and 10% conversion rate (100 paying users), revenue is $1,000/month. **Margins improve dramatically at scale** because fixed costs (auth, analytics, AI) don't grow linearly with users.

### Why This Differs from Traditional SaaS

Traditional CRUD apps are database-bound. Every user action requires a database write. Collaborative whiteboards keep active state in memory (Durable Objects, Redis) and only checkpoint to Postgres every 30-60 seconds.

Video/audio apps are bandwidth-bound. A 1-hour Zoom call consumes 500-900MB. A 1-hour collaborative whiteboard session consumes 5-10MB (mostly static assets on first load, then tiny WebSocket messages).

ML inference apps are compute-bound. Running Stable Diffusion costs $0.10-0.50 per image. Collaborative whiteboards offload compute to the client (Canvas rendering happens in the browser, not on servers).

## Technical Architecture: Why CRDT Overkill Lost to Simple Wins

### The Counterintuitive Finding: Most Production Whiteboards Avoid Full CRDTs

Figma: Custom per-property last-writer-wins with version numbers
Excalidraw: Version-number reconciliation with random nonce tie-breaking
tldraw: Server-authoritative sync with separate confirmed/pending layers

Only use CRDTs for: Rich text editing (Yjs in TipTap, ProseMirror, Quill)

### Why CRDTs Were Abandoned

Complexity vs. value trade-off. Full CRDT implementations (Yjs, Automerge) require 5,000-15,000 lines of code. They handle character-level conflict resolution for text. But whiteboards deal with discrete objects—sticky notes, shapes, connectors. The probability of two users simultaneously editing the **same property** of the **same object** is near zero.

Performance overhead. CRDTs maintain tombstones (deleted items) to ensure convergence. Over time, document size grows unbounded. Yjs documents can reach 10MB+ after thousands of edits, even if the visible content is 100KB. This breaks mobile clients with limited memory.

Developer experience cost. CRDTs require specialized knowledge. Debugging merge conflicts in CRDT documents is extremely difficult—the state can diverge in subtle ways that only appear after hundreds of operations.

### What Actually Works: Last-Writer-Wins with Version Numbers

The algorithm:
1. Every object property has a version number
2. On concurrent edits, highest version wins
3. Client generates random nonce for tie-breaking

Why this works:
- Simple to implement (50 lines of code)
- Predictable behavior (latest change always wins)
- Handles the 99.9% case (users don't edit same property simultaneously)
- Acceptable UX for the 0.1% case (one user's change gets overwritten, they just redo it)

When it fails: Two users simultaneously drag the same sticky note. One user's position update gets overwritten. This is rare enough (and low-stakes enough) that users tolerate it.

### Rendering Technology Choice Matters Enormously

Technology	Max Objects	Frame Rate	Best For
SVG	<1,000	30-60 FPS	Simple boards, easy debugging
Canvas 2D	1,000-10,000	60 FPS	Most use cases
WebGL	10,000-100,000+	60 FPS	Dense boards (Figma-level)

Why Canvas 2D is the default choice:

SVG hits DOM limits. Each object is a DOM node. Browsers slow down with 1,000+ nodes. Layout recalculations on every pan/zoom kill performance.

WebGL is overkill for 90% of boards. Requires shader knowledge, complex rendering pipeline, harder debugging. Only justified when boards regularly exceed 5,000 objects.

Canvas 2D balances performance and simplicity. Redraw the visible viewport every frame. Viewport culling (only render what's visible) handles 10,000+ objects easily.

## AI Integration: Why It Became Table Stakes in 18 Months

### The Timeline: Experimental → Expected in One Product Cycle

2024 Q1: Miro launches AI features as beta. Figma announces AI roadmap.
2024 Q3: tldraw's "Make Real" goes viral (hand-drawn → working code).
2025 Q1: Figma ships AI-generated prototypes at Config. Excalidraw adds AI diagram generation.
2026 Q1: AI features are baseline expectations, not differentiators.

### Why This Happened So Fast

Collaborative canvases are ideal AI interfaces. Unlike chat (linear, text-only), canvases are spatial and multimodal. You can draw a rough wireframe, point to it, and say "make this a working app." The AI's output appears directly on the canvas where you can iterate.

Structured output made it viable. GPT-4's function calling (2023) and structured outputs (2024) allow reliable generation of canvas elements. Before this, LLMs produced text that required parsing. Function calling returns JSON that maps directly to `createStickyNote(x, y, text)` operations.

The competitive moat shifted. Before AI, differentiation came from performance (Figma's WebGL rendering, Miro's sync infrastructure). These are commodity now—Cloudflare Durable Objects give everyone sub-100ms sync. AI features are the new moat.

### What AI Features Actually Cost

Miro's data: AI operations cost **$0.05-0.50 per user per month** in API fees. They use Azure OpenAI Service with structured prompts that return node/edge JSON.

Cost breakdown (per operation):
- Sticky note clustering: ~500 tokens input, 200 tokens output = $0.001 with GPT-4o-mini
- Diagram generation: ~1,000 tokens input, 2,000 tokens output = $0.004
- Board summarization: ~5,000 tokens input, 500 tokens output = $0.008

Why GPT-4o-mini is sufficient: 60% cheaper than GPT-3.5 Turbo, better performance for structured tasks. Whiteboard AI doesn't need reasoning—it needs reliable JSON generation.

### The Tiered Credit Strategy Prevents Runaway Costs

Problem: Users spam AI features, racking up $50+ in API costs per month.

Solution: Tiered limits:
- Free tier: 10 AI operations/month
- Pro tier ($10/month): 100 operations/month
- Team tier ($16/user/month): 500 operations/month

Why this works: Most users try AI features 5-10 times then stop. Power users (who actually need 100+ operations/month) are willing to pay. The credit system prevents $500 OpenAI bills from a single user running wild.

## Compliance Timeline: Why SOC 2 Waits Until $1M ARR

### The Certification Ladder Tied to Revenue Milestones

Day 1 (MVP): GDPR compliance (required for any EU user)
~$1M ARR: SOC 2 Type I ($10K-40K cost, 1-3 months)
6-12 months later: SOC 2 Type II ($30K-80K total)
$5M+ ARR: ISO 27001 (for international enterprise sales)
Only if healthcare-focused: HIPAA

### Why This Timeline Exists

GDPR on day 1 is non-negotiable. Cursor positions, presence data ("User X is viewing"), and activity logs all constitute personal data under GDPR. If you have a single EU user, GDPR applies. Penalties are 4% of global revenue.

SOC 2 at $1M ARR aligns with first enterprise deals. SMBs don't ask about compliance. Enterprises do. The $10K-25K deal threshold is where procurement asks for SOC 2 reports. Without it, you're eliminated from RFPs.

Type I → Type II requires 6-12 months. Type I is a point-in-time audit. Type II proves controls operated effectively over time. You can't skip Type I and go straight to Type II—auditors need the time-series data.

HIPAA is explicitly excluded by most products. Even Miro's Master Customer Agreement **prohibits** PHI submission. Why? HIPAA requires Business Associate Agreements, encryption at rest with customer-managed keys, detailed audit logs, and breach notification processes. The compliance burden is enormous relative to the healthcare market size for whiteboards.

### Why Data Residency Became Non-Negotiable for EU Enterprise

Miro: Offers EU data residency (Ireland primary, Germany backup) on all plans
Figma: EU hosting (Frankfurt/Dublin) is Enterprise-only, and only covers file content—billing metadata lives in US
MURAL: NA, EU, APAC regions available

Why this requirement emerged: GDPR's Schrems II ruling (2020) invalidated Privacy Shield. EU regulators now require data to stay in EU datacenters. Not offering EU residency eliminates you from enterprise shortlists in France, Germany, and Netherlands.

What "data residency" actually means: File content, edit history, and user activity must stay in EU. Metadata (billing, analytics) can live in US. Cursor positions during active sessions are exempt (real-time sync is considered "in transit").

## Performance Targets: Why <100ms Sync Latency Is Non-Negotiable

### The Industry-Standard Benchmarks

Metric	Target	Rationale
Edit sync latency	<100ms	Below human perception threshold
Cursor latency	<50ms	Smooth cursor tracking
Frame rate	60 FPS	Matches monitor refresh rate
Concurrent users	20-50 MVP, 200+ scale	Workshop/meeting sizes
Object capacity	5,000-10,000	Before viewport culling required

### Why These Numbers Matter

<100ms sync latency is the perception threshold. Below 100ms, changes feel instant. Above 150ms, users perceive lag. Above 300ms, the tool feels broken. This is why every major product targets sub-100ms.

How to achieve it: Five techniques must work together:
1. Optimistic local-first rendering (apply changes before server confirms)
2. Delta-based sync (send only changed properties, not full objects)
3. In-memory server state (no database round-trip for active documents)
4. Connection affinity (all users editing same board on same server)
5. Regional deployment (servers within ~1,500km of users)

<50ms cursor latency requires separation. Cursor updates must use a separate, lightweight channel. Cursor payloads are 20 bytes (`{userId, x, y}`). Sending them through the same channel as object updates (which can be 500+ bytes) causes queuing delays.

60 FPS is table stakes. Browsers render at 60 FPS. If your canvas renders at 30 FPS, panning feels janky. The solution: viewport culling (only render visible objects) and spatial indexing (quadtree lookups in O(log n) time instead of O(n) iteration).

### The Scaling Cliff: Why 200 Concurrent Users Is the Practical Limit

Miro's testing: 377 concurrent users on a single board before performance degraded. But this required dedicated server instances and was unusable in practice (lag, cursor jank).

Why 200 is the real limit:
- WebSocket message fan-out is $O(n^2)$. With 200 users, each edit generates 200 broadcasts. With 300 users, it's 300 broadcasts. This overwhelms network buffers.
- Browser memory limits. Each remote cursor consumes memory. At 300 cursors, browsers hit 500MB+ memory just for presence state.
- Cognitive overload. Humans can't track 300 cursors. Beyond 50 active users, the experience becomes chaotic.

Solution for larger sessions: Read-only viewers don't get write permissions. Only 10-20 "editors" can draw; 200+ "viewers" can watch. This reduces message fan-out by 10x.

## Strategic Takeaways for Your Team

### What Product-Market Fit Looks Like in This Category

Slow growth for years is normal. Miro took 7 years to reach 2M users. Figma took 6 years to reach 4M. The COVID spike was an anomaly, not the baseline.

The market is now mature and competitive. You're not competing against 2016 Figma (feature-poor, early market). You're competing against 2026 Miro (1,000+ templates, 50+ integrations, enterprise SSO).

Three viable business models exist:
1. Freemium → Enterprise (Miro path, requires $20M+ funding)
2. Developer SDK (tldraw path, requires strong open-source community)
3. Open-source + paid features (Excalidraw path, bootstrappable)

### What Technical Choices Actually Matter

Serverless wins for 0-10K concurrent users. Cloudflare Durable Objects offer the best cost/performance ratio. Self-managed WebSocket servers only make sense above 10K concurrent.

Full CRDTs are overkill. Last-writer-wins with version numbers handles 99.9% of cases. Only use Yjs if you're building rich text collaboration.

Canvas 2D hits the sweet spot. SVG is too slow above 1,000 objects. WebGL is too complex for most teams. Canvas 2D with viewport culling handles 10,000 objects easily.

AI features are mandatory, not optional. Every competitor ships AI diagram generation, sticky note clustering, and board summarization. Without these, you're perceived as outdated.

### What Cost Structure Enables

Infrastructure is cheap enough to ignore initially. At $100-260/month for 100 users, infrastructure cost is a rounding error compared to salaries. Optimize for development speed, not infrastructure cost.

Margins improve dramatically at scale. 10% conversion at $10/month pricing breaks even at 500 users ($500 revenue vs $500 infrastructure). At 5,000 users, margins hit 60% ($5,000 revenue vs $2,000 infrastructure).

The real costs are sales and customer acquisition. Collaborative tools have high CAC ($50-200 per customer) because they require team adoption. Infrastructure cost per user ($0.50/month) is negligible compared to CAC.

### What Compliance Unlocks

SOC 2 is the enterprise gate. Without it, you can't close deals above $10K/year. 60% of businesses prefer SOC 2-compliant vendors. A third of organizations have lost deals for lacking it.

EU data residency is table stakes for European enterprise. Not offering it eliminates you from RFPs in Germany, France, Netherlands. These represent 30-40% of the global enterprise whiteboard market.

HIPAA is a trap. The healthcare market is small for whiteboards. Compliance costs ($100K+) exceed revenue potential. Skip unless healthcare is your primary market.

# Phase 2: Architecture Discovery for LLM-Assisted Development

The most important constraint for architectural decisions: every component must work cohesively as a system while remaining straightforward for LLM coding agents to implement and maintain. This means favoring established patterns over novel architectures, choosing technologies with extensive documentation and examples, and prioritizing type safety to reduce runtime errors that LLM agents struggle to debug. The modern collaborative whiteboard stack has converged on a specific set of technologies that balance developer experience, performance, and maintainability.

## Hosting & Deployment: Serverless-First with Progressive Migration

The hosting landscape for real-time collaborative applications has transformed dramatically. **Vercel cannot serve WebSocket backends** — its serverless functions have a 10-second execution limit incompatible with persistent connections. This architectural constraint has pushed the industry toward three distinct deployment models, each optimized for different growth stages.

### Recommended Deployment Strategy by Scale

0–1,000 concurrent users: Cloudflare Workers + Durable Objects represents the optimal starting point. tldraw runs its entire production infrastructure on this architecture, serving hundreds of thousands of sessions. Each collaborative room gets a dedicated Durable Object instance that maintains in-memory state and scales automatically across Cloudflare's global network. Cost structure: $5/month base + ~$0.15 per million requests + $0.02/GB-month for Durable Objects storage. A room with 50 concurrent users costs approximately $0.10/month with hibernation. Total infrastructure for 1,000 concurrent users: $50–$150/month.

The critical advantage for LLM-assisted development: Cloudflare Workers use standard Web APIs (Fetch, WebSocket, Request/Response), making them immediately familiar to AI coding agents trained on web standards. The wrangler CLI provides straightforward local development (`wrangler dev`) and deployment (`wrangler deploy`), which fits naturally into AI-generated workflows.

1,000–10,000 concurrent users: Hybrid architecture with Railway or Render. At this scale, dedicated WebSocket servers become cost-effective. Railway pricing: $20/vCPU/month + $10/GB RAM/month, with customers reporting 40–75% cost savings versus AWS. A 4vCPU/8GB instance handles ~5,000 concurrent WebSocket connections at $140/month. Combine this with Cloudflare Workers for static asset delivery and API routes. Total infrastructure: $400–$1,200/month.

Railway and Render both offer GitHub integration with automatic deployments, Dockerfile support, and environment variable management — all concepts that LLM agents handle well through standardized patterns. The `railway.json` or `render.yaml` configuration files provide declarative infrastructure-as-code that AI can generate and modify reliably.

10,000+ concurrent users: Dedicated infrastructure on AWS/GCP with regional deployment. At enterprise scale, committed use discounts (40–72% savings) and reserved instances justify the operational complexity. Figma runs dedicated Rust processes for each active document, with PostgreSQL for metadata, DynamoDB for write-ahead logging, and S3 for assets. This tier requires DevOps expertise but achieves $0.02–$0.05 per concurrent user per month at steady state.

### CI/CD Requirements for Real-Time Applications

The unique challenge for collaborative whiteboards: **zero-downtime deployments while maintaining active WebSocket connections.** Standard blue-green deployments break long-lived connections. The production-tested pattern:

1. **Graceful shutdown with connection draining:** New connections route to the updated deployment, while existing connections remain on the old version for up to 30 minutes
2. **Database migrations run before deployment** with backward-compatible schema changes (additive only, never destructive)
3. **Feature flags control rollout** of new multiplayer sync protocol changes
4. **Automated testing must include WebSocket integration tests** — not just HTTP API tests

GitHub Actions workflows for collaborative apps typically include:
- TypeScript compilation and type-checking (catches 80% of bugs before deployment)
- End-to-end WebSocket connection tests with Playwright
- Load testing with Artillery or k6 (simulate 100–1,000 concurrent connections)
- Automatic deployment to preview environments for every pull request
- Canary deployments to 5% of production traffic before full rollout

For LLM agents, this CI/CD complexity is managed through well-documented GitHub Actions templates (e.g., Vercel's Next.js templates, tldraw's Cloudflare deployment workflows). AI coding assistants excel at adapting these established patterns rather than creating novel deployment pipelines.

### Scaling Characteristics: When and How to Migrate

The critical inflection points:

100 → 1,000 concurrent users: Move from Cloudflare's pay-per-use to reserved Durable Objects capacity. Enable hibernatable WebSocket connections to reduce costs 10x.

1,000 → 5,000 concurrent users: Add dedicated WebSocket servers on Railway/Render behind Cloudflare load balancing. Implement connection affinity (sticky sessions) so all clients editing the same document connect to the same server instance.

5,000 → 10,000 concurrent users: Horizontal scaling with Redis Pub/Sub for cross-server message broadcasting. Each WebSocket server subscribes to document-specific Redis channels. Requires 2-5ms added latency but enables unlimited horizontal scaling.

10,000+ concurrent users: Regional deployments with geo-routing. Deploy WebSocket servers in 3+ regions (US-East, EU-West, Asia-Pacific) and route users to the nearest region. Requires inter-region replication for global teams editing the same document — typically solved with Cloudflare's global network or AWS Global Accelerator.

## Authentication & Authorization: Managed Auth Wins

The authentication landscape for Next.js applications in 2025 has three clear tiers, each with distinct trade-offs for LLM-assisted development.

### Recommended Authentication Stack

Clerk: Best for LLM-assisted development and rapid iteration. Clerk provides pre-built UI components (`<SignIn />`, `<SignUp />`, `<UserButton />`), automatic session management, and webhook-based database syncing. Setup time: 7 minutes from empty project to working authentication. Free tier: 10,000 monthly active users. Paid tier: $25/month + $0.02/MAU after 10,000.

Why Clerk excels for AI-generated code: The SDK is designed for type safety with full TypeScript support. LLM agents can import Clerk hooks (`useUser()`, `useAuth()`) and immediately get IntelliSense-driven autocomplete for session data. The declarative component model (`<SignedIn>`, `<SignedOut>`) maps directly to React patterns that AI coding assistants already understand well. Example:

```typescript
import { SignedIn, SignedOut, UserButton } from "@clerk/nextjs";

export default function Header() {
  return (
    <header>
      <SignedOut>
        <SignInButton />
      </SignedOut>
      <SignedIn>
        <UserButton />
      </SignedIn>
    </header>
  );
}
```

NextAuth.js v5: Best for cost-conscious projects with time to invest. Open-source, zero per-user pricing, complete data ownership. Setup complexity: 1–3 hours including custom UI development. LLM agents can generate NextAuth configurations, but debugging authentication edge cases (token refresh, session persistence across tabs) requires human oversight.

The key advantage: NextAuth v5's universal `auth()` function works across Server Components, API routes, and middleware — a single pattern that AI agents can apply consistently. However, multi-tenancy and organization management require custom implementation (20–40 hours of development time).

Auth0: Best for enterprise compliance requirements only. SOC 2, HIPAA, ISO 27001 certifications. Pricing: $23–$240+ per month + $0.05/MAU. The extensive compliance portfolio justifies the cost premium only when selling to regulated industries. Setup complexity: medium (30–60 minutes).

Supabase Auth: Best when already using Supabase for database. Free tier: 50,000 MAU (5x Clerk's free tier). Built-in Row-Level Security integration means permission checks happen at database layer. Setup complexity: low (15–30 minutes).

Why it makes sense: You're already using Supabase for PostgreSQL—adding auth eliminates a vendor (no separate Clerk subscription). RLS policies reference `auth.uid()` directly, putting authorization logic in the database where data lives. Supports magic links, OAuth providers, email/password.

Trade-offs: No pre-built UI components like Clerk's `<SignIn />` (build login forms yourself). No organizations feature out of the box (requires custom implementation). Less polished DX—more configuration needed.

When to choose: Comfortable building auth UI, want maximum free tier (50K vs 10K MAU), prefer database-layer authorization, value vendor consolidation.

When to skip: Need organizations/multi-tenancy from day one (Clerk's API is superior), want zero auth configuration (Clerk's components are plug-and-play), need advanced features like device tracking.

### RBAC and Multi-Tenancy Patterns

Collaborative whiteboards require three permission levels:

1. **Board-level permissions:** View, Edit, Comment, Admin
2. **Organization-level permissions:** Member, Workspace Admin, Owner
3. **Feature-level permissions:** AI tools, Export, Integrations (for tiered pricing)

Clerk's Organizations API provides built-in multi-tenancy with custom roles definable through the dashboard. The permission check pattern that LLM agents generate reliably:

```typescript
import { auth } from "@clerk/nextjs";

export async function canEditBoard(boardId: string) {
  const { userId, orgRole } = auth();
  if (!userId) return false;

  // Check board-specific permissions in database
  const permission = await db.boardPermissions.findFirst({
    where: { boardId, userId }
  });

  return permission?.role === "editor" || permission?.role === "admin" || orgRole ===
"org:admin";
}
```

For NextAuth implementations, Row-Level Security (RLS) in PostgreSQL provides database-native authorization. Supabase excels here with automatic RLS policy generation. Example policy that AI agents can generate:

```sql
CREATE POLICY "Users can edit boards they own or are members of"
ON boards FOR UPDATE
USING (
  auth.uid() = owner_id OR
  auth.uid() IN (
    SELECT user_id FROM board_members
    WHERE board_id = boards.id AND role IN ('editor', 'admin')
  )
);
```

### Multi-Tenancy Considerations

Three architectural patterns for multi-tenant collaborative whiteboards:

Row-based tenancy (recommended for MVP): Single database with `organization_id` column on every table. Simplest to implement, works well up to 10,000 organizations. Every query includes `WHERE organization_id = ?`. LLM agents handle this pattern reliably through Prisma schema definitions:

```prisma
model Board {
  id             String   @id @default(cuid())
  organizationId String
  organization   Organization @relation(fields: [organizationId], references: [id])

  @@index([organizationId])
}
```

Schema-based tenancy: Separate PostgreSQL schema per organization. Better isolation, enables per-tenant backups, but adds complexity for cross-tenant features (e.g., public template gallery). Requires manual schema management that's error-prone for LLM-generated migrations.

Database-per-tenant: Complete isolation, required for some enterprise compliance scenarios. Operational overhead is substantial — not recommended unless contractually required.

## Database & Data Layer: PostgreSQL + Redis is the Standard

Every successful collaborative whiteboard uses the same database architecture with minor variations. The pattern is so consistent that LLM agents can scaffold it reliably from examples.

### Core Database Architecture

PostgreSQL (or PostgreSQL-compatible) for all persistent data:
- User accounts, organizations, permissions
- Board metadata (name, owner, creation date, settings)
- Sharing links, comments, version history
- Subscription and billing data

Recommended hosted options:
- **Supabase:** Free tier includes 500MB database + 1GB file storage + 2GB data transfer. Excellent for MVP. Paid: $25/month for 8GB database. Built-in Row-Level Security, automatic API generation, real-time subscriptions. LLM agents work well with Supabase's JavaScript SDK.
- **Neon:** Serverless PostgreSQL with automatic scaling to zero. Free tier: 0.5GB storage, 3GB data transfer. Paid: $19/month for 10GB. Branch-based development (like Git for databases) — each pull request gets an isolated database copy. Exceptional for LLM-assisted development workflows.
- **PlanetScale:** MySQL-compatible, horizontal sharding built-in. Free tier: 5GB storage, 1 billion row reads/month. Paid: $39/month for 100GB. Schema changes via pull requests with

zero-downtime migrations. Best for massive scale (10M+ users), but adds MySQL-specific quirks that confuse some LLM agents.

Redis (or compatible) for ephemeral real-time state:
- Active WebSocket connections per document
- User presence (who's online, cursor positions)
- Pub/Sub channels for broadcasting updates across multiple servers
- Rate limiting and session caching

Recommended options:
- **Upstash Redis:** Serverless with per-request pricing. Free tier: 10,000 commands/day. Paid: $0.20 per 100,000 commands. REST API means it works from serverless functions. Ideal for Cloudflare Workers deployments.
- **Redis Cloud:** Free tier: 30MB. Paid: $5/month for 100MB, $15/month for 500MB. Standard Redis protocol, full feature set.

### Real-Time Sync: CRDTs vs. Operational Transform vs. Last-Writer-Wins

The surprising research finding: **most production collaborative whiteboards do NOT use full CRDT implementations.**

Last-writer-wins with version numbers (used by Figma, Excalidraw): Each object property has a version number. Concurrent edits to the same property use the highest version number. Simple, fast, works well for discrete objects where simultaneous edits to the exact same property are rare. LLM agents generate this pattern reliably:

```typescript
interface BoardObject {
  id: string;
  x: number;
  y: number;
  version: number; // Increments with each update
  lastModified: Date;
  lastModifiedBy: string;
}

function mergeUpdates(local: BoardObject, remote: BoardObject): BoardObject {
  return {
    ...local,
    x: remote.version > local.version ? remote.x : local.x,
    y: remote.version > local.version ? remote.y : local.y,
    version: Math.max(local.version, remote.version) + 1
  };
}
```

Yjs CRDT (recommended for rich text editing only): When your whiteboard includes text-heavy features (markdown notes, collaborative documents), Yjs provides character-level conflict resolution. Yjs integrates with TipTap, ProseMirror, Quill, Monaco — all editors that LLM agents can configure from documentation.

Yjs persistence to PostgreSQL pattern (AI agents can scaffold this from examples):

```typescript
import * as Y from 'yjs';
import { PostgresqlPersistence } from 'y-postgresql';

const ydoc = new Y.Doc();
const persistence = new PostgresqlPersistence(connectionString);

// Load document from database
const update = await persistence.getYDoc(documentId);
if (update) Y.applyUpdate(ydoc, update);

// Save updates to database
ydoc.on('update', async (update) => {
  await persistence.storeUpdate(documentId, update);
});
```

tldraw's sync protocol (best for building on top of existing SDKs): tldraw provides a complete sync solution (`@tldraw/sync`) that handles document state, presence, and persistence. If you're building a whiteboard from scratch with tldraw's SDK, their sync protocol is production-proven and LLM-friendly.

### Read/Write Ratios and Caching Strategy

Collaborative whiteboards have inverted read/write ratios compared to traditional web apps:

- **During active editing:** 80% writes, 20% reads. Users are creating/moving objects constantly.
- **During viewing:** 95% reads, 5% writes. Teams review boards in meetings.

Caching strategy for LLM-generated backends:

1. **Never cache active board state** — defeats the purpose of real-time collaboration
2. **Cache board thumbnails aggressively** (24-hour CDN TTL) — used in board gallery views
3. **Cache user profile data** (5-minute TTL) — username, avatar URL rarely change

4. **Cache permission checks** (30-second TTL in Redis) — reduces database load during active editing

Example caching pattern that AI agents reliably generate:

```typescript
async function getBoardPermission(userId: string, boardId: string) {
  const cacheKey = `perm:${userId}:${boardId}`;

  // Try cache first
  const cached = await redis.get(cacheKey);
  if (cached) return JSON.parse(cached);

  // Cache miss - query database
  const permission = await db.permissions.findFirst({
    where: { userId, boardId }
  });

  // Cache for 30 seconds
  await redis.setex(cacheKey, 30, JSON.stringify(permission));
  return permission;
}
```

## Backend/API Architecture: tRPC for Internal APIs

The most LLM-friendly backend architecture for TypeScript collaborative whiteboards: **tRPC with Next.js App Router.** This combination eliminates API contracts entirely — frontend and backend share types automatically through TypeScript's type system.

### Why tRPC Dominates for LLM-Assisted Development

Traditional REST APIs require three separate code artifacts that drift out of sync:
1. Backend route definitions
2. Request/response TypeScript interfaces
3. Frontend API client code

LLM agents struggle with keeping these synchronized. A backend change requires updating all three files, and AI often misses edge cases in the manual type mappings.

tRPC collapses this to a single source of truth:

```typescript
// Backend procedure (server/routers/board.ts)
```

```
import { router, protectedProcedure } from '../trpc';
import { z } from 'zod';

export const boardRouter = router({
  getBoard: protectedProcedure
    .input(z.object({ boardId: z.string() }))
    .query(async ({ input, ctx }) => {
      const board = await ctx.db.board.findUnique({
        where: { id: input.boardId },
        include: { objects: true }
      });
      if (!board) throw new Error('Board not found');
      return board; // TypeScript infers return type automatically
    }),
});

// Frontend usage (components/BoardView.tsx)
import { trpc } from '@/utils/trpc';

export function BoardView({ boardId }: { boardId: string }) {
  const { data: board, isLoading } = trpc.board.getBoard.useQuery({ boardId });

  if (isLoading) return <LoadingSpinner />;
  // `board` is fully typed - AI gets autocomplete for board.objects, board.name, etc.
  return <Canvas objects={board.objects} />;
}
```

If the backend changes `board` to include a new field like `board.template`, the frontend immediately gets that type without any manual intervention. LLM agents can refactor backends safely — TypeScript compilation fails if frontend code breaks.

### REST vs. GraphQL vs. tRPC Decision Matrix

Use tRPC when:
- Building an internal API (not exposing to third parties)
- Full-stack TypeScript codebase (frontend and backend in same repo)
- Team uses Next.js or similar TypeScript framework
- Priority is development velocity and type safety

Use REST when:
- Public API that external developers will consume
- Multi-language teams (mobile app in Swift, web app in TypeScript, backend in Go)
- Need long-term API stability with versioning (v1, v2, v3 endpoints)

- Webhooks for third-party integrations (Stripe, Slack)

Use GraphQL when:
- Complex data requirements with deep nesting (querying boards → objects → comments → replies)
- Mobile clients on slow networks (query only needed fields to reduce bandwidth)
- Multiple client types with different data needs
- Willing to invest 2–4 weeks in GraphQL infrastructure setup

For collaborative whiteboards specifically: **tRPC handles 95% of use cases.** The remaining 5% are public APIs (share board publicly without auth) which should be separate REST endpoints anyway. Example architecture:

- Internal API (authenticated users, collaborative editing): tRPC
- Public API (embed board in website, share read-only view): Next.js API routes with REST
- Webhooks (notify external systems when board changes): Next.js API routes

### Background Jobs and Queue Requirements

Real-time collaborative apps need background processing for:

1. **Thumbnail generation:** When a board updates, regenerate its preview image asynchronously
2. **Export to PDF/PNG:** Large boards take 5–30 seconds to render
3. **AI operations:** Multi-step LLM workflows (analyze board → generate suggestions → create elements)
4. **Email notifications:** Digest emails for board activity
5. **Data cleanup:** Delete inactive boards after 90 days

Recommended background job architecture:

0–1,000 users: Vercel Cron Jobs + Upstash QStash
- Vercel cron runs scheduled tasks (hourly cleanup, daily digest emails)
- QStash provides HTTP-based queue for async jobs
- Cost: Free tier handles 10,000 jobs/month
- LLM agents can generate cron configs in `vercel.json`

1,000–10,000 users: BullMQ with Redis
- Industry-standard job queue for Node.js
- Handles retries, delays, rate limiting automatically
- Cost: Included in Redis hosting (~$15/month for 500MB Redis)
- LLM agents generate worker patterns from BullMQ documentation

10,000+ users: Inngest or Temporal

- Durable execution with visual workflow DAGs
- Built-in retries, observability, and distributed tracing
- Cost: Inngest $200/month, Temporal $300/month
- More complex for AI agents but handles enterprise reliability requirements

## Frontend Framework & Rendering: Next.js App Router with React Server Components

The frontend architecture that every major collaborative whiteboard either uses or migrates toward: **Next.js 14+ with App Router and React Server Components.**

### Why Next.js Dominates

Next.js provides four capabilities that collaborative whiteboards require:

1. **Server-side rendering (SSR) for SEO:** Public board links need Open Graph previews for Slack/Twitter unfurling
2. **React Server Components:** Fetch permissions and board metadata on the server, render UI without client-side waterfall requests
3. **API routes:** tRPC procedures run as Next.js API routes — single deployment for frontend and backend
4. **Edge runtime:** Deploy API routes to Cloudflare/Vercel Edge for <50ms global latency

LLM coding agents excel with Next.js because:
- Clear file-system routing conventions (`app/boards/[id]/page.tsx` → `/boards/:id`)
- Standardized patterns for data fetching (`async function BoardPage({ params })`)
- Extensive documentation and Stack Overflow examples for agent training data

### SEO Requirements and Server Rendering Strategy

Collaborative whiteboards need SEO for:
- **Public board links:** When a user shares a board publicly, search engines should index it
- **Template galleries:** Browse 500+ templates by category, all SEO-discoverable
- **Landing pages:** Marketing pages for feature descriptions, pricing, use cases

Rendering strategy by page type:

Static generation (build time) for:
- Marketing pages, pricing, documentation
- Template gallery pages (regenerate daily with `revalidate: 86400`)
- Cost: Zero — served from CDN

Server-side rendering (request time) for:
- Public board preview pages (generate Open Graph meta tags dynamically)
- User dashboard (personalized, permission-aware)

- Cost: ~1-2ms per request on Vercel Edge

Client-side rendering (no server rendering) for:
- The collaborative canvas itself — requires WebSocket connections, real-time cursor tracking
- Board editing UI — too stateful for server rendering benefits
- Cost: Zero server cost, runs entirely in browser

Example Next.js page structure that LLM agents generate correctly:

```typescript
// app/boards/[id]/page.tsx (Server Component)
import { BoardCanvas } from '@/components/BoardCanvas';
import { auth } from '@clerk/nextjs';
import { db } from '@/lib/db';

export async function generateMetadata({ params }) {
  const board = await db.board.findUnique({ where: { id: params.id } });
  return {
    title: board.name,
    openGraph: { images: [board.thumbnailUrl] }
  };
}

export default async function BoardPage({ params }) {
  const { userId } = auth();
  const board = await db.board.findUnique({ where: { id: params.id } });

  // Server-rendered permission check
  if (!board.isPublic && board.ownerId !== userId) {
    return <AccessDenied />;
  }

  // Client Component for real-time collaboration
  return <BoardCanvas boardId={params.id} initialData={board} />;
}
```

### Offline Support and PWA Considerations

Most collaborative whiteboards do NOT implement full offline support. Why? The complexity-to-value ratio is unfavorable:

Offline editing requires:
1. IndexedDB for local storage (5-10MB board data)

2. Service worker for offline assets
3. Conflict resolution when reconnecting
4. UI indicators for offline state
5. Background sync for queued changes

Alternative approach (recommended): "Optimistic UI" without true offline persistence. Changes render immediately in the browser, queue in memory, sync when online. If the tab closes before syncing, changes are lost — but users understand this tradeoff.

LLM agents can implement optimistic UI reliably:

```typescript
const [objects, setObjects] = useState(initialObjects);
const [pendingChanges, setPendingChanges] = useState([]);

function updateObject(id, changes) {
  // Update UI immediately
  setObjects(prev => prev.map(obj =>
    obj.id === id ? { ...obj, ...changes } : obj
  ));

  // Queue for sync
  setPendingChanges(prev => [...prev, { id, changes }]);

  // Sync to server
  syncToServer({ id, changes })
    .then(() => setPendingChanges(prev => prev.filter(c => c.id !== id)))
    .catch(() => toast.error('Failed to sync - changes lost'));
}
```

Full PWA implementation only justifies the cost when:
- Selling to industries with intermittent connectivity (aviation, construction)
- Mobile-first product (tablets used in field work)
- Compliance requires data never leave the device

## Third-Party Integrations: Stripe, Resend, PostHog, OpenAI

Modern collaborative whiteboards integrate 4-8 third-party services. Each service category has a clear winner for LLM-assisted development based on API design quality and documentation clarity.

### Payments: Stripe is Non-Negotiable

Stripe handles 99% of SaaS payment needs: subscriptions, usage-based billing, invoices, tax compliance, dunning emails. The API is exceptionally well-documented, making it LLM-friendly. Free tier: unlimited test mode. Production pricing: 2.9% + $0.30 per transaction + $0.005 per ACH transaction.

Key integration points for collaborative whiteboards:

```typescript
// Create checkout session for board subscription
const session = await stripe.checkout.sessions.create({
  mode: 'subscription',
  line_items: [{ price: 'price_1234', quantity: 1 }],
  customer_email: user.email,
  success_url: `${baseUrl}/boards?session_id={CHECKOUT_SESSION_ID}`,
  cancel_url: `${baseUrl}/pricing`,
});

// Webhook handler for subscription events
export async function POST(req: Request) {
  const event = stripe.webhooks.constructEvent(
    await req.text(),
    req.headers.get('stripe-signature'),
    process.env.STRIPE_WEBHOOK_SECRET
  );

  if (event.type === 'customer.subscription.updated') {
    await db.user.update({
      where: { stripeCustomerId: event.data.object.customer },
      data: { subscriptionTier: event.data.object.items.data[0].price.id }
    });
  }
}
```

LLM agents generate Stripe integration code accurately because:
1. Stripe's official TypeScript SDK has excellent type definitions
2. Stripe documentation includes copy-paste code examples for every use case
3. The Vercel Next.js Subscription Payments template (7.6K GitHub stars) provides a reference implementation

Critical pricing consideration: Stripe's 2.9% + $0.30 fee structure means **monthly subscriptions below $10 lose 30–40% to transaction fees.** Solution: Offer annual subscriptions with 20% discount — improves unit economics and reduces churn.

### Email: Resend for Transactional + Marketing

Resend dominates for LLM-friendly email infrastructure. Free tier: 3,000 emails/month. Paid: $20/month for 50,000 emails. The API uses React components for email templates, which AI agents already understand:

```typescript
import { Resend } from 'resend';
import { BoardInviteEmail } from '@/emails/BoardInvite';

const resend = new Resend(process.env.RESEND_API_KEY);

await resend.emails.send({
  from: 'CollabBoard <noreply@collabboard.app>',
  to: invitee.email,
  subject: `${inviter.name} invited you to collaborate`,
  react: <BoardInviteEmail boardName={board.name} inviterName={inviter.name} />
});
```

Alternative consideration: **Loops.so** for marketing automation (drip campaigns, newsletters). $50/month for 5,000 contacts. Better for non-technical founders — visual email builder instead of code templates.

### Analytics: PostHog for Product Analytics

PostHog provides the full analytics stack: event tracking, session replay, feature flags, A/B testing, SQL access to raw data. Open-source core (self-hostable) or cloud. Free tier: 1 million events/month, 5,000 session replays/month. Paid: $0.00031/event after free tier (~$300/month for 1 billion events).

Integration with Next.js App Router (both server and client):

```typescript
// app/providers.tsx (Client Component)
'use client';
import posthog from 'posthog-js';
import { PostHogProvider } from 'posthog-js/react';

if (typeof window !== 'undefined') {
  posthog.init(process.env.NEXT_PUBLIC_POSTHOG_KEY, {
    api_host: 'https://app.posthog.com',
  });
}
```

```
export function Providers({ children }) {
  return <PostHogProvider client={posthog}>{children}</PostHogProvider>;
}

// Server-side feature flags
import { PostHog } from 'posthog-node';

const posthog = new PostHog(process.env.POSTHOG_API_KEY);
const flags = await posthog.getAllFlags(userId);

if (flags['new-ai-features']) {
  return <AIFeatures />;
}
```

Alternative: **Vercel Analytics** for page views and Core Web Vitals ($10/month for 100K events). Simpler but lacks feature flags and session replay.

Critical finding: **Free analytics tiers (Plausible, Umami, Google Analytics) lack the product analytics depth needed for SaaS.** You need to track events like "User invited collaborator," "Board exported to PDF," "AI feature used" — not just page views. PostHog's generous free tier makes it the default choice.

### AI: OpenAI with Structured Outputs

OpenAI API for AI features (board summarization, diagram generation, smart layouts). Pricing: GPT-4o $2.50/1M input tokens, $10/1M output tokens. GPT-4o-mini $0.15/1M input tokens, $0.60/1M output tokens — **60% cheaper than GPT-3.5 Turbo with better performance.**

Recommended AI architecture pattern:

```typescript
import OpenAI from 'openai';

const openai = new OpenAI({ apiKey: process.env.OPENAI_API_KEY });

const completion = await openai.chat.completions.create({
  model: 'gpt-4o-mini',
  messages: [
    { role: 'system', content: 'You are a whiteboard assistant. Generate structured JSON output for creating board elements.' },
    { role: 'user', content: `Create a SWOT analysis template with 4 quadrants` }
  ],
  response_format: { type: 'json_object' }
});

const elements = JSON.parse(completion.choices[0].message.content);
```

Cost optimization: **Use tiered credit systems.** Free tier: 10 AI operations/month. Pro tier: 100 operations/month. Prevents runaway API costs from users spamming AI features. Store per-user AI operation counts in Redis with monthly reset.

Vendor lock-in mitigation: Abstract AI calls behind your own interface:

```typescript
interface AIProvider {
  generateDiagram(prompt: string): Promise<BoardElement[]>;
}

class OpenAIProvider implements AIProvider {
  async generateDiagram(prompt: string) {
    // OpenAI implementation
  }
}

class AnthropicProvider implements AIProvider {
  async generateDiagram(prompt: string) {
    // Claude implementation
  }
}
```

LLM agents can swap providers without touching business logic.

### Service Integration Cost Summary (1,000 Users Baseline)

Service	Free Tier	Paid Tier	Monthly Cost @ 1K Users	Notes
Stripe	Unlimited test	2.9% + $0.30	$290 (10% pay $10/mo)	Revenue share, not fixed cost
Resend	3K emails	$20/mo for 50K	$20	Transactional emails only
PostHog	1M events	$0.00031/event	$100	~100K events/user/month
OpenAI	$5 credit	Pay-per-token	$50	10 operations/user/month
Clerk	10K MAU	$25 + $0.02/MAU	$25	Using free tier
Cloudflare	100K req/day	$5/mo + usage	$50	Workers + R2 + Durable Objects

| Supabase | 500MB DB | $25/mo | $25 | Database + auth + storage |
| **Total** | | | **$560/month** | At 1,000 monthly active users |

Revenue requirement to break even: $560 / 1,000 users = $0.56 per user per month. With 10% conversion to $10/month paid tier: $1 revenue per user. **Margins improve dramatically at scale** as free tiers exhaust and percentage-based costs (Stripe 2.9%) become the dominant expense.

### Pricing Cliffs and Rate Limits

Critical thresholds that cause cost spikes:

1. **Clerk: 10,000 → 10,001 MAU:** Jump from $0 to $25/month + $0.02/MAU. At 15,000 MAU: $25 + (5,000 × $0.02) = $125/month
2. **PostHog: 1M → 1.01M events/month:** First million free, then $0.00031/event. At 10M events: $2,790/month
3. **Supabase: 500MB → 8GB database:** Free tier ends, jump to $25/month. At 50GB: $125/month
4. **Resend: 3,000 → 3,001 emails/month:** Jump from free to $20/month

Mitigation strategy: **Monitor usage dashboards weekly, set up billing alerts at 80% of free tier limits.** When approaching cliff, either optimize usage (reduce event volume, archive old data) or graduate to paid tier with growth funding secured.

Rate limits to design around:

- **OpenAI:** 10,000 requests/minute on Tier 1 ($100+ spent). Below that: 500 requests/minute. Implement client-side rate limiting and queuing.
- **Stripe API:** 100 requests/second in test mode, 25/second in live mode (below $1M processing). Use Stripe webhook events instead of polling API.
- **Cloudflare Workers:** 1,000 requests/minute on free plan. Burst to 10,000/minute, then throttle.

LLM agents can generate rate-limiting middleware from examples:

```typescript
import { Ratelimit } from '@upstash/ratelimit';
import { Redis } from '@upstash/redis';

const ratelimit = new Ratelimit({
  redis: Redis.fromEnv(),
  limiter: Ratelimit.slidingWindow(10, '10 s'), // 10 requests per 10 seconds
});
```

```
export async function POST(req: Request) {
  const { userId } = auth();
  const { success } = await ratelimit.limit(userId);

  if (!success) {
    return new Response('Rate limit exceeded', { status: 429 });
  }

  // Process request
}
```

## Cohesive System Architecture: The Recommended Stack

Based on comprehensive analysis of production collaborative whiteboards and LLM agent capabilities, the optimal architecture for a team of one developer supported by AI coding agents:

Frontend: Next.js 14+ App Router + React + TypeScript + Tailwind CSS + shadcn/ui
Backend: tRPC + Next.js API Routes + Zod validation
Database: Supabase (PostgreSQL + auth + storage) or Neon (serverless Postgres)
Real-time: Cloudflare Durable Objects (via tldraw sync protocol) or custom WebSocket server on Railway
Canvas Rendering: tldraw SDK (if building on existing solution) or HTML5 Canvas with Rough.js (if building from scratch)
Authentication: Clerk (rapid iteration) or NextAuth.js v5 (cost-conscious)
Payments: Stripe Checkout + Webhooks
Email: Resend with React Email templates
Analytics: PostHog (events + feature flags + session replay)
AI: OpenAI GPT-4o-mini with structured outputs
Deployment: Vercel (frontend + API routes) + Cloudflare Workers (WebSocket backend)
CI/CD: GitHub Actions with Vercel automatic deployments

Why this stack excels for LLM-assisted development:

1. **Type safety end-to-end:** TypeScript from database (Prisma schema) through backend (tRPC procedures) to frontend (React components). LLM agents get instant feedback on type errors.

2. **Declarative patterns:** Most configuration is declarative files (`schema.prisma`, `wrangler.toml`, `vercel.json`) that AI agents parse and modify reliably.

3. **Extensive documentation:** Every component has official documentation, tutorial videos, and thousands of GitHub examples. AI training data includes these resources.

4. **Progressive complexity:** Start with Vercel's one-click deploy, add complexity (multi-region, background jobs, advanced caching) only when metrics justify it.

5. **Cost-effective scaling:** $50–$150/month at launch, $400–$1,200/month at 1,000 concurrent users, $2,500–$6,800/month at 10,000 concurrent users.

This architecture powers tldraw (500K MAU), Excalidraw (850K MAU), and dozens of Y Combinator collaborative tools. The patterns are proven, the costs are predictable, and LLM coding agents can scaffold 80% of the implementation from established templates.

# Phase 3: Post-Stack Refinement for CollabBoard

Stack: Next.js 14 App Router + tRPC + TypeScript + Cloudflare Durable Objects + Clerk + Supabase + Stripe + PostHog + OpenAI

## Security Vulnerabilities: Why This Stack Has Unique Risks

### The Next.js Environment Variable Problem

Why it matters: Variables prefixed `NEXT_PUBLIC_` get embedded in JavaScript bundles sent to browsers. Developers intuitively prefix any variable "needed on the client," but this creates permanent exposure—even after rotating secrets, old bundles in CDN caches still contain them.

The reasoning: Traditional backend frameworks force environment variable isolation by default. Next.js breaks this assumption with the public prefix convention. For LLM-assisted development, this is particularly dangerous because AI agents don't understand the security implications of naming conventions—they'll happily suggest `NEXT_PUBLIC_STRIPE_SECRET_KEY` if the prompt mentions "frontend needs Stripe."

Decision: Audit every environment variable. Only analytics IDs and truly public values (like Clerk publishable keys, which are designed for client exposure) should use the prefix.

### Server Components Serialize Everything

Why it matters: React Server Components automatically serialize props passed to Client Components. If you pass an entire Prisma user object, it serializes fields like `email`, `stripeCustomerId`, and `hashedPassword` to the client bundle—even if the UI only displays the name.

The reasoning: Traditional SPAs use explicit API calls where you choose return values. RSC collapses this boundary, making it invisible where server-only data ends and client data begins. The default behavior is "serialize everything," which is the opposite of secure-by-default.

Decision: Never pass database records directly to Client Components. Always destructure or use Prisma `select` to create explicit allowlists of safe fields.

### tRPC's Type Safety Creates a False Sense of Security

Why it matters: Type safety guarantees shape, not content validity. An attacker can send `{ boardId: "../../../etc/passwd" }` and TypeScript accepts it as `string`. Without Zod validation, tRPC procedures become direct database access for any client.

The reasoning: Traditional REST APIs force validation at the HTTP boundary because request bodies are untyped. tRPC eliminates this forcing function. LLM agents frequently skip `.input()` calls because the procedure still type-checks without them.

Decision: Every tRPC procedure must use `.input(z.object(...))`. Make this a pre-commit hook that fails without it.

### Supabase RLS vs Application-Layer Authorization

Why it matters: If RLS isn't enabled on a table, the service role key (which your backend uses) bypasses all security. This is fundamentally different from ORMs like Prisma where authorization is application logic—here, the database IS the security boundary.

The reasoning: Supabase's design philosophy is "database-first authorization." This works brilliantly when configured correctly but fails catastrophically when misconfigured. A single table without RLS creates a complete security bypass.

Decision: Enable RLS on every table immediately, even for MVP. Test policies by connecting as different users. Never use the service role key on the client (Supabase's anon key is designed for client use because RLS protects it).

### WebSocket State Persistence in Durable Objects

Why it matters: Unlike stateless functions that reset after each invocation, Durable Objects maintain in-memory state indefinitely. An attacker can open 10,000 WebSocket connections to a single room and exhaust memory, crashing the entire collaborative session.

The reasoning: Traditional WebSocket servers have process-level limits (max connections per process) enforced by operating systems. Durable Objects are V8 isolates with soft limits—they'll keep accepting connections until memory is exhausted.

Decision: Implement per-room connection limits (e.g., max 300 connections) and per-connection message rate limits (60 messages/second). Use hibernatable WebSockets to reduce memory footprint.

### Webhook Signature Verification

Why it matters: Clerk and Stripe send webhooks (HTTP POST requests) when events occur. Without signature verification, attackers can forge webhooks claiming a user upgraded to Pro, granting unauthorized access.

The reasoning: Webhooks are the only way to stay synchronized with external services, but they're public HTTP endpoints. The signature proves the request came from Clerk/Stripe and wasn't tampered with.

Decision: Always verify webhook signatures before processing events. Use Clerk's Svix library and Stripe's constructEvent method—don't implement signature verification manually.

### Dependency Risks in This Stack

Why specific dependencies are high-risk:

- **Next.js:** Major versions (14→15) break App Router patterns. Canary releases contain experimental features that become production defaults. Pin to exact minor versions (`14.2.18` not `^14.0.0`).
- **tRPC:** v10→v11 changed the entire API surface. Requires manual migration for every procedure.
- **Clerk:** Session management implementation changes between SDK versions. Test authentication flows after every update.
- **Prisma:** The generated client changes with schema migrations. If you deploy without regenerating the client, runtime errors occur.

The reasoning: This stack is younger than established frameworks (Express, Django). Breaking changes happen frequently, and the ecosystem hasn't stabilized. For LLM-assisted development, this is dangerous—AI agents trained on v10 patterns will generate broken code for v11.

Decision: Use exact versions for critical dependencies. Run `npm audit` weekly. Enable Dependabot with auto-merge for patch versions only.

## File Structure: Why Monorepo Wins for This Stack

### Monorepo vs Polyrepo Decision

Why monorepo is optimal:

1. **Type sharing is the killer feature.** tRPC depends on importing backend types in frontend code (`import type { AppRouter }`). In a polyrepo, you'd publish types to npm, adding 5-10 second latency between backend changes and frontend updates.

2. **Shared Zod schemas eliminate duplication.** WebSocket message validation and tRPC input validation use identical schemas. Monorepo lets you define once, import everywhere.

3. **Single CI/CD pipeline.** Cloudflare Workers backend and Next.js frontend deploy together. Version mismatches between deployed services cause runtime errors that don't appear in type checking.

4. **LLM agents struggle with multi-repo coordination.** AI coding assistants can't easily navigate between repositories. Monorepos keep all context in a single workspace.

Why polyrepo would be better (but doesn't apply here):

- Multiple teams with different tech stacks (mobile team using Swift, web team using TypeScript)
- Backend serves multiple frontends with different deployment schedules
- Need independent versioning and release cycles

Decision: Monorepo with pnpm workspaces. Add Turborepo for build caching (10x faster CI after initial run).

### Feature-Based vs Layer-Based Organization

Why hybrid approach is optimal:

Components: Feature-based. Grouping `BoardCanvas.tsx`, `BoardToolbar.tsx`, `useBoardSync.ts` in a `board/` folder makes it obvious what code relates to the board feature. When an LLM agent is asked "modify the board toolbar," it knows exactly where to look.

Backend: Layer-based. Grouping all tRPC routers together makes it clear what API surface exists. Services layer contains reusable business logic that multiple routers call. This prevents duplication when `board.ts` and `ai.ts` both need permission checking.

The reasoning: Frontend components are high-cohesion (tightly related files change together). Backend code is high-coupling (many routers use the same services). Different organization patterns serve different change patterns.

Decision: Feature-based for frontend, layer-based for backend. Monorepo structure places frontend and backend in separate `apps/` to make the distinction obvious.

### Shared Packages Strategy

Why three packages:

1. **`database` package:** Contains Prisma schema. Both Next.js and Cloudflare Workers import the generated Prisma client. Prevents schema drift between services.

2. **`schemas` package:** Zod schemas for tRPC inputs and WebSocket messages. Both frontend (for optimistic validation) and backend (for security) use identical schemas.

3. **`types` package:** Shared TypeScript types that aren't tied to Prisma or Zod. Example: `type BoardPermission = 'view' | 'edit' | 'admin'`.

The reasoning: Shared code enforces consistency. Without shared schemas, frontend and backend validation can diverge (frontend accepts 100-character board names, backend accepts 50). LLM agents can't maintain consistency across duplicated code.

Decision: Extract shared code early, even for MVP. The monorepo tooling overhead is minimal with pnpm workspaces.

## Naming Conventions: Why Consistency Matters for LLM Agents

### The Cognitive Load Problem

Why naming conventions matter more with AI assistance: Human developers can adapt to inconsistent naming because they understand context. LLM agents pattern-match on naming conventions. If 80% of files use kebab-case but 20% use camelCase, the AI doesn't know which to generate.

The reasoning: AI coding assistants are trained on GitHub's public repositories, which have wildly inconsistent conventions. Establishing strict conventions in YOUR codebase gives the AI a strong local signal to override its training data.

### File Naming: Why kebab-case

Decision: `board-canvas.tsx`, `use-board-sync.ts`, `format-date.ts`

Reasoning:
- Works on all operating systems (Windows is case-insensitive for files)
- Matches URL conventions (`/boards/board-123`)
- Next.js file-based routing uses kebab-case by convention

- Easier to type than PascalCase (no shift key required)

Exception: Components use PascalCase because they must match the exported React component name (\`BoardCanvas.tsx\` exports \`BoardCanvas\`). This is a forced consistency by TypeScript.

### Variable Naming: Why camelCase

Decision: `const boardId`, `function fetchBoardData()`

Reasoning: JavaScript community standard since ES5. TypeScript inherits this convention. All major style guides (Airbnb, Google, Standard) mandate camelCase for variables.

### tRPC Procedure Naming: Why Verb Prefixes

Decision: `getBoard`, `listBoards`, `createBoard`, `updateBoard`, `deleteBoard`

Reasoning: Verb prefixes make the operation explicit. Without prefixes, `board()` is ambiguous—does it get or create? Verb prefixes map to HTTP methods (GET, POST, PUT/PATCH, DELETE) but remain framework-agnostic.

Alternative considered: REST-style naming (`board.find`, `board.findMany`, `board.create`). Rejected because Prisma already uses this pattern for database operations—using it for API methods creates confusion.

### Why ESLint + Prettier Over Manual Enforcement

The problem: Humans forget to follow conventions. LLM agents will follow whatever pattern exists in the current file.

The solution: Automated enforcement. ESLint fails the build if conventions are violated. Prettier auto-fixes formatting on save. Pre-commit hooks prevent non-compliant code from entering the repository.

Reasoning: For LLM-assisted development, this is critical. AI agents don't read style guides—they read the actual code. If a human manually violates conventions once, the AI will replicate that violation in every subsequent file it generates.

Decision: Configure ESLint + Prettier immediately. Add pre-commit hooks that run both tools. Make violations CI failures.

## Testing Strategy: Why Coverage Matters Less Than You Think

### The MVP Coverage Paradox

Why 40-50% is the right target for MVP: Higher coverage requires exponentially more effort. Going from 40% to 80% coverage takes 3-4x as much time as writing the initial 40%. For a one-week MVP, that's the difference between shipping and not shipping.

The reasoning: Early-stage startups die from lack of product-market fit, not bugs. Users tolerate bugs if the core value proposition works. They won't tolerate a product that ships three months late.

What to actually test for MVP:
- Authentication flows (Clerk integration)
- Payment processing (Stripe webhooks)
- Core collaborative editing (WebSocket sync)
- Critical security paths (authorization checks)

What to skip for MVP:
- UI component rendering tests (visual testing is expensive)
- Edge cases in non-critical features
- Performance benchmarks
- Accessibility compliance

Decision: 40-50% coverage for MVP, with 80%+ coverage on critical paths (auth, payments, permissions). Increase to 70-80% overall after securing Series A funding.

### Vitest vs Jest: Why Speed Matters

The decision: Use Vitest for all unit and integration tests.

Reasoning:
- **5-10x faster** for TypeScript projects. Jest uses `ts-jest` which transpiles on every run. Vitest uses esbuild, which is 100x faster than TypeScript's compiler.
- **Native ESM support.** Next.js 13+ uses ES modules. Jest requires complex configuration to handle ESM. Vitest works out of the box.
- **Watch mode performance.** During development, tests re-run on file changes. Jest watch mode can take 5-10 seconds. Vitest reruns in under 1 second.

Why this matters for LLM agents: AI coding assistants generate tests based on examples. Vitest's API is Jest-compatible, so the AI can use the vast corpus of Jest examples from its training data. You get speed without retraining the AI.

Alternative considered: Jest is more established with better documentation and Stack Overflow coverage. Rejected because developer experience matters more than documentation breadth—tests that run 10x faster get run 10x more often.

### Playwright vs Cypress: Why Multi-Browser Testing Matters

The decision: Use Playwright for end-to-end tests.

Reasoning:
- **Parallel execution built-in.** Playwright runs tests across multiple workers by default. Cypress requires paid tier for parallelization.
- **Multi-browser support.** Playwright tests Chromium, Firefox, and WebKit in a single command. Collaborative whiteboards render differently across browsers (Canvas API quirks, WebSocket behavior).
- **WebSocket testing first-class.** Playwright can intercept WebSocket frames. Testing real-time collaboration requires this capability.
- **TypeScript-native.** Configuration and test files are TypeScript. Cypress requires additional setup for TypeScript support.

Why this matters for real-time collaboration: The critical path is "two users see each other's changes." This requires spinning up two browser contexts, connecting to the same WebSocket, and verifying message propagation. Playwright's API makes this straightforward; Cypress makes it painful.

Alternative considered: Cypress has better documentation and a larger community. The component testing feature is excellent. Rejected because real-time WebSocket testing is non-negotiable for a collaborative whiteboard.

### Mocking Philosophy: Mock External Services, Not Your Own Code

Why mock Clerk, Stripe, OpenAI:
- External services have rate limits (OpenAI: 500 req/min on Tier 1)
- Tests shouldn't depend on network connectivity
- External APIs have unpredictable latency (Stripe webhooks can take 30 seconds)
- Cost accumulation (OpenAI charges per token, even in test mode)

Why NOT to mock Supabase or Prisma:
- You're testing YOUR authorization logic in RLS policies
- Database queries are the critical path—mocking them tests nothing
- Prisma has an in-memory SQLite option for fast tests
- Supabase provides test database provisioning

The reasoning: Mock the things you don't control (external APIs). Test the things you do control (database logic, authorization).

Decision: Use mock libraries (vitest-mock-extended) for external services. Use real databases (SQLite in-memory for unit tests, Postgres for integration tests) for data access.

## Recommended Tooling: Why These Specific Tools

### VS Code Extensions: The Essential Three

1. ESLint + Prettier (tied together): Show errors inline before commit. For LLM-assisted development, this means the AI sees linting errors immediately when generating code, allowing it to self-correct.

2. Error Lens: Displays TypeScript errors inline in the editor (not just in the Problems panel). This prevents the "it compiles locally but fails in CI" problem—developers see errors where they're typed, not 30 lines down.

3. Tailwind CSS IntelliSense: Autocompletes Tailwind classes and shows the generated CSS on hover. Without this, developers have to memorize class names or constantly reference documentation. LLM agents don't use this extension (they're trained on Tailwind docs), but humans need it.

Why not more extensions: Every extension slows down the editor. TypeScript language server already consumes significant CPU. Adding 20 extensions makes VS Code sluggish, degrading the developer experience.

Decision: Recommend 5-7 essential extensions maximum. Document them in `.vscode/extensions.json` so new developers get prompted to install them.

### CLI Tools: Why Global vs Local Installation

Install globally:
- `pnpm` (package manager)
- `wrangler` (Cloudflare deployments)
- `tsx` (run TypeScript files without compilation)

Install locally (in project):
- `prisma` (database migrations)
- `eslint` (linting)
- `vitest` (testing)

The reasoning: Global tools are stable and version-agnostic. You want the same `pnpm` across all projects. Local tools have project-specific configurations that break across versions. Different projects use different ESLint rules; installing globally creates version conflicts.

Decision: Global tools for developer workflows, local tools for project-specific behavior.

### Debugging Next.js: Why Server Component Debugging Is Hard

The problem: Server Components run on the server. Client Components run in the browser. A single page render executes code in two separate JavaScript runtimes. Traditional breakpoints don't work across this boundary.

The solution: Use separate debug configurations for server and client. VS Code's launch.json can attach to both simultaneously, but developers must understand which code runs where.

Why this matters for LLM agents: AI coding assistants often generate Server Components when Client Components are needed (and vice versa). The only reliable way to catch this is seeing console.log output in the wrong place—server logs appear in terminal, client logs in browser console.

Decision: Configure dual debugging from day one. Add clear comments marking Server vs Client Components so AI agents can distinguish them.

### Turbo for Monorepo Builds: Why Caching Matters

The problem: In a monorepo, changing one file potentially requires rebuilding all packages. Without caching, CI re-runs the entire build pipeline (5-10 minutes) for every commit.

The solution: Turbo caches build outputs based on input hash. If nothing changed in the `database` package, Turbo skips its build and uses cached output. This reduces CI time from 8 minutes to 2 minutes (5x speedup).

Why this matters for LLM-assisted development: AI agents make many small commits as they iterate. Fast CI feedback loops let you catch errors quickly. Slow CI (8+ minutes) creates a deployment bottleneck where multiple commits queue up, making debugging harder.

Decision: Add Turbo to monorepo from day one. The configuration overhead is minimal (single `turbo.json` file) and the speedup is immediate.

## Summary: The Reasoning Behind Each Decision

Security: This stack collapses traditional boundaries (RSC serializes everything, tRPC eliminates API validation points, Supabase makes the database the authorization layer). Every security decision compensates for these collapsed boundaries.

File Structure: Monorepo enables type sharing, which is the entire value proposition of tRPC. Feature-based frontend organization aids LLM navigation; layer-based backend prevents code duplication.

Naming Conventions: Consistency matters more with AI assistance because LLMs pattern-match. Automated enforcement (ESLint/Prettier) prevents drift.

Testing: Coverage targets balance shipping speed against quality. 40-50% for MVP is optimal; 70-80% for post-PMF. Vitest for speed, Playwright for real-time WebSocket testing.

Tooling: Minimize extensions to avoid editor slowdown. Cache builds with Turbo to keep CI fast. Debug configurations handle Next.js's split runtime architecture.

Each decision optimizes for LLM-assisted development: type safety, automated enforcement, clear conventions, fast feedback loops.

# Phase 4: Building a production-grade AI agent for CollabBoard

The single highest-impact change for CollabBoard's AI agent is shifting from raw LLM coordinate calculation to a plan-then-execute architecture with deterministic layout tools. This pattern — validated by tldraw's Agent Starter Kit, Figma AI, and Miro — lets the LLM decide *what* to create and *which* layout strategy to use, while server-side algorithms handle exact positioning. Combined with model routing (GPT-4o-mini for simple commands, GPT-4.1 for complex spatial work), streaming progressive rendering via Vercel AI SDK's `streamText()`, and Inngest for serverless concurrency, CollabBoard can achieve reliable spatial output, sub-$2,000/month costs at 10K MAU, and a fluid real-time UX. What follows is a deep technical blueprint across all eight research areas.

## 1. Model routing cuts costs by 85% without sacrificing quality

### The pricing landscape

Model pricing has evolved dramatically. The key models for CollabBoard's tool-calling workload, with per-1M-token rates:

Model	Input	Output	Cached Input	Sweet Spot
GPT-4o-mini	$0.15	$0.60	$0.075 (50% off)	Simple create/delete/move

GPT-4.1-mini	$0.40	$1.60	$0.10 (75% off)	Balanced tool calling
GPT-4.1	$2.00	$8.00	$0.50 (75% off)	Complex spatial reasoning
Claude Sonnet 4	$3.00	$15.00	$0.30 (90% off)	Highest multi-step reliability
GPT-4.1-nano	$0.10	$0.40	$0.025 (75% off)	Routing classifier

GPT-4.1-mini deserves special attention: it has a **1M-token context window**, excels specifically at tool calling per OpenAI's benchmarks, and costs only 2.7× GPT-4o-mini — a compelling middle ground that didn't exist when CollabBoard was first architected.

### How to implement routing

The RouteLLM framework (Berkeley/LMSYS, ICLR 2025) demonstrated **95% of GPT-4 quality using only 26% GPT-4 calls**, yielding ~48% cost reduction. For CollabBoard, a three-tier routing approach works best:

Tier 1 — Rule-based pre-classifier (0ms, free): Pattern-match against the user command. Single-object operations (`"create a sticky note"`, `"delete the arrow"`, `"change color to blue"`) route to GPT-4o-mini. Multi-object spatial commands (`"arrange in a grid"`, `"create a flowchart"`, `"organize these notes"`) route to GPT-4.1.

Tier 2 — Ambiguous commands: When regex patterns don't match, use GPT-4.1-nano (~$0.00003 per classification) to output a single token: `simple`, `complex`. This adds ~100ms latency but handles edge cases reliably.

Tier 3 — Progressive escalation: If the cheap model's output fails validation (overlapping objects, out-of-bounds), automatically re-run with GPT-4.1. This catches the ~5% of "simple" commands that actually need stronger reasoning.

### Cost projections at scale

At **10,000 MAU × 20 commands/session × 4 sessions/month = 800,000 commands/month** (70% simple, 30% complex):

Strategy	Monthly Cost	Quality
GPT-4o-mini only	**$325**	Struggles with complex layouts
GPT-4.1-mini only	**$867**	Strong tool calling, good balance
Hybrid: 4o-mini + GPT-4.1	**$3,000**	High quality where it matters
Hybrid + prompt caching + semantic cache	**$1,450–1,650**	Same quality, 45-55% savings
Claude Sonnet 4 only	**$7,560**	Premium quality, very expensive

Prompt caching alone saves 30-40% on input costs — OpenAI's automatic caching (zero configuration) applies a 50-75% discount on cached prefixes ≥1,024 tokens. Since

CollabBoard's system prompt + tool definitions likely total 1,500-2,000 tokens, the majority of every request is cacheable. Anthropic's explicit caching offers **90% savings** on cached reads but requires manual `cache_control` breakpoints.

### OpenAI vs Anthropic for tool calling

For this specific use case, **OpenAI models are the stronger choice** for three reasons: (1) first-class Vercel AI SDK support as the primary provider, (2) GPT-4.1-mini's tool-calling benchmarks specifically outperform alternatives at its price point, and (3) the 75% cached-input discount on GPT-4.1 family models is more aggressive than Anthropic's pricing for light-to-medium workloads. That said, Claude Sonnet 4's "think" tool — which lets the model reason explicitly between tool calls in long chains — produces noticeably more reliable multi-step operations. The recommendation: **GPT-4o-mini/4.1-mini as default, with Claude Sonnet 4 as an option for users who need premium spatial reasoning** (or as the complex-command model in the routing tier).

## 2. Let the LLM choose layouts, not coordinates

The most critical architectural insight from studying production AI canvas tools: **LLMs should never calculate individual pixel coordinates for complex layouts.** Research consistently shows LLM spatial reasoning accuracy degrades **42-80%** as compositional complexity increases. Every production tool that works well has arrived at the same pattern.

### What tldraw, Figma, and Miro actually do

tldraw's Agent Starter Kit (the closest analog to CollabBoard) uses a dual-context approach: the agent receives both a viewport screenshot AND structured shape data, then outputs high-level actions like `create`, `align`, `distribute`, and `stack`. Critically, `align` and `distribute` are **built-in actions that call tldraw's native layout APIs** — the LLM never calculates alignment coordinates. The system prompt provides concrete size hints: *"shapes are 50×50, text is 24pt tall, each character is ~12px wide."*

Figma AI generates designs using Figma's auto-layout system (constraints, padding, responsive components), not raw coordinates. The LLM outputs a structured component tree with relative relationships that Figma's engine resolves into absolute positions.

Miro AI generates *content* (sticky note text, diagram descriptions) and delegates all spatial arrangement to Miro's built-in auto-layout and clustering algorithms.

### Layout tools to expose to the AI

The optimal architecture: the LLM decides *what* to place and *which* layout strategy to use; deterministic algorithms calculate *where*.

```typescript
// Tools the AI should call instead of calculating coordinates:
arrangeInGrid(objectIds, columns, spacing, startPosition)
alignObjects(objectIds, 'left' | 'center' | 'right' | 'top' | 'middle' | 'bottom')
distributeEvenly(objectIds, 'horizontal' | 'vertical', spacing?)
stackObjects(objectIds, 'horizontal' | 'vertical', gap)
layoutAsTree(rootId, edges, direction: 'TB' | 'LR', spacing)
placeRelativeTo(referenceId, position: 'above' | 'below' | 'left' | 'right', offset)
```

For graph/tree layouts, **dagre** (simple directed graph layout, ~5KB) and **ELK.js** (full-featured with layered, force, stress, radial algorithms) are the standard libraries. Both integrate cleanly with server-side Node.js execution.

### Prompting techniques that actually improve spatial output

When the LLM *must* calculate coordinates (simple 1-3 element placements), three techniques have strong empirical backing:

Chain-of-Symbol (CoS) prompting achieved a **60.8% accuracy gain** on spatial tasks by replacing natural language descriptions with condensed symbolic representations. Instead of "the red block is on top of the blue block, which is to the left of the green block," use `red ↑ blue ← green`.

JSON Cartesian coordinates dramatically outperform natural language or ASCII-grid spatial representations in LLM benchmarks. Representing board state as `{id, type, x, y, width, height}` JSON arrays aligns with training data distributions.

Explicit few-shot coordinate calculations in the system prompt:
```
User: Place 3 boxes in a row, each 100x80, 20px spacing, starting at (50,50)
Step-by-step: Box1: x=50, y=50. Box2: x=50+100+20=170, y=50. Box3: x=170+100+20=290, y=50.
```

A system prompt should declare the coordinate system explicitly (*"origin top-left, x increases right, y increases down, positions are top-left corner of shapes"*), provide the viewport bounds, and include size references for standard elements.

## 3. Plan-then-execute is the right multi-step pattern

Three patterns exist for multi-step AI commands. For whiteboard operations, one clearly dominates.

### Why plan-then-execute wins

The **plan-then-execute** pattern separates AI reasoning from side effects: the LLM generates a complete JSON plan describing all objects, positions, and connections; deterministic code validates and executes it. This offers five advantages over the alternatives:

1. **Validation before mutation** — the entire plan can be checked for overlaps, out-of-bounds elements, and spacing uniformity before any object is created
2. **Atomicity** — if validation fails, nothing is persisted; no partial state to clean up
3. **Security** — tool outputs during execution can't alter the plan (immune to prompt injection via tool results)
4. **Cost efficiency** — planning uses one LLM call; execution is pure code
5. **Predictability** — no surprise tool calls or runaway loops

The Vercel AI SDK 6 supports this natively through `Output.object()` with Zod schemas:

```typescript
const WhiteboardPlanSchema = z.object({
  objects: z.array(z.object({
    type: z.enum(['rectangle', 'ellipse', 'sticky-note', 'text', 'arrow']),
    id: z.string(),
    x: z.number(), y: z.number(),
    width: z.number(), height: z.number(),
    text: z.string().optional(),
    style: z.object({ fill: z.string().optional() }).optional(),
  })),
  connections: z.array(z.object({
    fromId: z.string(), toId: z.string(), type: z.enum(['arrow', 'line']),
  })).optional(),
  layout: z.object({
    strategy: z.enum(['grid', 'flowchart', 'freeform', 'tree', 'mindmap']),
  }).optional(),
});

const { output: plan } = await generateText({
  model: 'openai/gpt-4.1',
  output: Output.object({ schema: WhiteboardPlanSchema }),
  system: WHITEBOARD_SYSTEM_PROMPT,
  prompt: userCommand,
```

});

### Self-correction with a two-retry ceiling

When validation fails, feed the specific errors back to the LLM for re-planning (maximum 2 retries). If the plan still fails after retries, apply **deterministic auto-correction**: clamp objects to viewport bounds, nudge overlapping elements apart, and snap to the nearest grid point. This hybrid approach — LLM reasoning + programmatic fallback — handles the long tail of edge cases that pure self-correction misses.

### Partial failure handling

For plan-then-execute, use **all-or-nothing transactions** via Supabase/Postgres. The plan is validated before execution, so runtime failures should be rare. Wrap execution in a Postgres transaction:

```sql
BEGIN;
INSERT INTO board_objects (board_id, type, x, y, ...) VALUES ...;  -- all objects
INSERT INTO connectors (board_id, from_id, to_id, ...) VALUES ...;  -- all connections
COMMIT;
```

For independent operations (e.g., "add labels to these 5 shapes"), use **keep-what-succeeded + report failures** — track each operation individually and report partial results to the user.

## 4. Stream objects onto the canvas as they're created

### Progressive rendering is the right UX

Three UX approaches exist for AI canvas operations: optimistic UI (ghost placeholders), wait-for-confirmation (loading → batch render), and progressive rendering (objects appear one by one). **Progressive rendering** is the clear winner for canvas apps — each shape appears as the actual final result with a brief fade-in animation, creating an engaging "AI painting on canvas" experience. No placeholder mismatch, no jarring batch appearance.

The Vercel AI SDK enables this through `streamText()` with `createUIMessageStream` and custom data parts:

```typescript
const stream = createUIMessageStream({
  execute: ({ writer }) => {
    writer.write({
      type: 'data-aiStatus',
      data: { status: 'planning', message: 'AI is analyzing your request...' },
      transient: true, // Not stored in message history
    });

    const result = streamText({
      model: 'openai/gpt-4.1',
      tools: canvasTools,
      stopWhen: stepCountIs(10),
      experimental_onToolCallFinish({ toolName, output, success }) {
        if (success) {
          writer.write({
            type: 'data-objectCreated',
            data: { toolName, object: output },
          });
        }
      },
    });
    writer.merge(result.toUIMessageStream());
  },
});
```

Key limitation discovered: tool *results* are not incrementally streamed within a single tool execution — the `execute()` function runs to completion before the result is sent. But between sequential tool calls in a multi-step operation, each completed tool result streams immediately to the client.

### Animation and feedback patterns

- **Staggered appearance**: 50-100ms delay between objects appearing, with fade-in + scale (0.8→1.0 over 200ms)
- **Contextual status**: Show what the AI is doing ("Planning layout…", "Creating shapes…", "Connecting elements…") near the user's cursor, not as a global spinner
- **Single undo unit**: Group all AI-created objects into one undo step so Ctrl+Z reverts the entire operation
- **Broadcast individually**: Send each created object to other users via Supabase Realtime Broadcast as it completes, so collaborators see progressive updates too

```typescript
// Server-side: broadcast each tool result to other users
experimental_onToolCallFinish({ toolName, output, success }) {
  if (success) {
    await supabaseAdmin.from('board_objects').insert(output);
    await supabaseAdmin.channel(`canvas:${roomId}`).send({
      type: 'broadcast',
      event: 'ai-object-created',
      payload: { userId, shape: output, toolName },
    });
  }
}
```

## 5. Tiered context with semantic search for large boards

### Three-tier context architecture

For boards with 500+ objects, naively serializing everything consumes ~50K tokens. The proven pattern is spatial windowing with hierarchical detail:

Tier 1 — Viewport objects (full detail): Complete properties for the ~50-100 objects currently visible. These are the objects the user is most likely referencing.

Tier 2 — Nearby objects (compact): Objects within 1-2 viewport widths, serialized as `{id, type, text_preview, bbox, color}`. Enough for the LLM to reference them without consuming excessive tokens.

Tier 3 — Board summary (statistical): "Board contains 342 additional objects: 89 sticky notes clustered in 3 groups, 45 rectangles, 23 connectors." Cluster distant objects spatially using simple k-means on position coordinates.

This mirrors tldraw's Agent Starter Kit exactly, which categorizes shapes into `FocusedShape` (full properties), `BlurryShape` (bounds/id/type/text only), and `PeripheralShapeCluster` (grouped counts with approximate position).

### Embeddings for semantic object retrieval

When a user says *"group the research notes"*, the system needs to find objects with research-related text. Supabase's pgvector extension handles this elegantly:

```sql
CREATE EXTENSION IF NOT EXISTS vector;
ALTER TABLE board_objects ADD COLUMN embedding halfvec(1536);
CREATE INDEX ON board_objects USING hnsw (embedding halfvec_cosine_ops);
```

~~~

Use `text-embedding-3-small` at **$0.02 per 1M tokens** — embedding a 500-object board costs ~$0.0005. Generate embeddings asynchronously on text content changes, then query with cosine similarity. This enables a hybrid resolution pipeline for natural language object references:

1. **Parse reference** → LLM extracts `{visual_filters: {color: "blue", type: "rectangle"}, semantic_query: "marketing"}`
2. **Structured filter** → SQL query on type, color, size properties
3. **Semantic narrow** → pgvector similarity search on the filtered set
4. **Temporal/deictic** → "that thing I just created" resolves via recent actions log; "it" resolves via anaphora from conversation context

---

6. Inngest solves the serverless concurrency problem

The current per-user promise chain via a module-level `Map` fails across multiple Vercel serverless instances — each instance has its own memory space, so concurrent requests from the same user can bypass the queue entirely.

Inngest is purpose-built for this

Inngest provides per-user concurrency control that works across all serverless instances, with built-in retries, idempotency, and an observability dashboard:

```typescript
export const processWhiteboardCommand = inngest.createFunction(
  {
    id: 'process-whiteboard-command',
    concurrency: [{
      limit: 1,
      key: 'event.data.userId', // Per-user serial execution
    }],
    retries: 2,
  },
  { event: 'whiteboard/command.received' },
  async ({ event, step }) => {
    const plan = await step.run('generate-plan', () =>
      generateValidatedPlan(event.data.command)
    );
    await step.run('execute-plan', () =>
      executePlanInTransaction(plan, event.data.boardId)
~~~

    );
  }
);
~~~

**Inngest advantages**: native Vercel Marketplace integration, **100K free executions/month**, FIFO ordering per concurrency key, each `step.run()` is individually retryable and durable. The alternative — **Upstash Redis Lock** (`@upstash/lock`) — works for simple mutual exclusion with ~1ms latency but lacks Inngest's workflow orchestration capabilities.

Avoid Postgres advisory locks for AI operations — they hold a database connection for the entire AI generation duration (5-30 seconds), which can exhaust Supabase's connection pool through PgBouncer.

### Validation heuristics before persisting

Run these checks on every AI-generated plan before execution:

- **Viewport bounds**: all objects within viewport (auto-fix: clamp to bounds)
- **Overlap detection**: AABB collision check with minimum 10-20px gap (auto-fix: nudge apart)
- **Grid spacing uniformity**: max deviation < 15px across rows/columns
- **Connector validity**: all `fromId`/`toId` reference existing objects
- **Text fitting**: estimated text height (lines × fontSize × 1.4 + padding) fits within container height
- **Minimum size**: no objects smaller than 20×20px
- **Object count**: warn if plan creates > 50 objects

---

## 7. Vision adds genuine value for spatial understanding

### When to send a screenshot

Sending a canvas screenshot to GPT-4o or Claude adds **~85 tokens at low detail** ($0.0002 per request) — negligible cost. The value is highest when the user's command references visual layout that structured data poorly captures: *"make this look more balanced"*, *"the section on the left is too crowded"*, *"add a title above this group"*.

tldraw's Agent Starter Kit uses a **dual-context approach**: both a viewport screenshot AND structured shape data. The screenshot provides spatial gestalt; the structured data provides precise coordinates and properties. For CollabBoard, implement this as an optional enhancement:

```typescript
~~~

const screenshot = canvas.toDataURL('image/png', 0.5); // 50% quality
// Send as: { type: 'image_url', image_url: { url: screenshot, detail: 'low' } }
```

Use `detail: 'low'` (fixed 85 tokens) for routine operations. Reserve `detail: 'high'` (~765-1,105 tokens for 1024-2048px images) for explicitly visual commands.

Multi-turn conversation context

Replace the current 5-minute TTL anaphora resolution with a proper **sliding window + summarization** approach:

- **Short-term**: Last 3-5 complete conversation turns (verbatim, ~2,000-4,000 tokens) stored in Upstash Redis with 15-30 minute TTL
- **Medium-term**: Rolling summary of the session, updated every 3-5 turns using GPT-4o-mini (~$0.001 per compression), capturing session intent, mentioned objects, and actions taken
- **Long-term**: Key facts about user preferences stored in Supabase with embeddings for cross-session retrieval

Multi-user AI conflict resolution

For concurrent AI operations on the same board, the simplest effective pattern is **per-board AI mutex** via Redis — only one AI operation runs at a time per board, with a 30-second TTL. This prevents conflicting spatial modifications. For better UX, implement **region-based locking**: AI commands lock only the spatial regions containing affected objects, allowing concurrent AI operations on different parts of the board.

Model AI outputs as the same property-level operations used by the existing Supabase Realtime sync. Figma's approach — **per-property, last-writer-wins** — works well since two users rarely modify the exact same property of the same object simultaneously.

---

8. Prioritized implementation roadmap

Ordered by **impact-to-effort ratio**, with estimated implementation time:

Phase 1 — Foundational improvements (1-2 weeks)

1. **Switch to plan-then-execute with structured output** — Replace the current single-shot `generateText()` with `Output.object()` + Zod schema validation. Add bounds-checking, overlap detection, and auto-correction. *This alone will dramatically improve layout reliability.* (3-4 days)
```

2. **Add layout engine tools** — Implement `arrangeInGrid`, `alignObjects`, `distributeEvenly`, and `stackObjects` as server-side functions exposed as AI tools. Remove the burden of coordinate math from the LLM. (2-3 days)

3. **Fix serverless concurrency** — Replace the module-level `Map` with Inngest (or Upstash Redis Lock as a quick fix). This prevents race conditions that corrupt board state. (1-2 days)

### Phase 2 — Quality and UX (2-3 weeks)

4. **Implement model routing** — Rule-based pre-classifier + GPT-4.1-nano fallback. Route simple commands to GPT-4o-mini, complex spatial commands to GPT-4.1. Enable prompt caching. Expected cost reduction: **45-55%** from the complex-model baseline. (2-3 days)

5. **Stream tool results progressively** — Switch from `generateText()` to `streamText()` with `createUIMessageStream` and custom data parts. Show objects appearing one by one with fade-in animations. Add contextual "AI is working" status indicators. (3-4 days)

6. **Upgrade context management** — Implement three-tier spatial windowing (viewport/nearby/summary). Update system prompt with coordinate system declaration, size references, and viewport bounds. (2-3 days)

### Phase 3 — Advanced features (3-4 weeks)

7. **Semantic search with pgvector** — Add embeddings column to board objects, create HNSW index, implement the hybrid reference resolution pipeline (structured filter → semantic search → temporal context). (3-4 days)

8. **Multi-turn session memory** — Replace 5-minute TTL anaphora with sliding window + summarization via Upstash Redis. Enable genuine multi-turn reasoning across commands. (2-3 days)

9. **Vision context** — Add optional screenshot capture sent alongside structured data for spatial-understanding commands. Implement at `detail: 'low'` (85 tokens) by default. (1-2 days)

10. **Collaborative AI conflict resolution** — Implement per-board AI mutex with region-based locking for concurrent operations. Broadcast AI status to all board participants. (3-4 days)

### What to skip (for now)

- **Fine-tuning GPT-4o-mini** on whiteboard tool calls — only worthwhile after collecting 5K+ labeled examples from production usage
- **Full CRDT integration for AI operations** — last-writer-wins at the property level is sufficient until concurrent AI usage becomes frequent

- **Semantic caching** — implement only after reaching >5,000 MAU when the 20-40% cache hit rate produces meaningful savings

The first three items — plan-then-execute, layout tools, and concurrency fix — will produce the largest quality improvement for approximately one sprint of work. Model routing and streaming are the second-highest-value investments, directly improving both cost efficiency and perceived responsiveness.

# Phase 5: # Securing real-time collaborative apps against six critical vulnerability classes

Real-time collaborative applications — whiteboards, shared canvases, document editors — face a unique security surface where **latency constraints, multi-user state synchronization, and complex permission models** intersect. The most dangerous vulnerabilities in these systems are not exotic cryptographic failures but rather architectural blind spots: tokens that aren't scoped to resources, mutation handlers that forget to check permissions, and debug endpoints that ship to production. This report synthesizes guidance from OWASP, Figma's engineering team, the ShareDB and Hocuspocus open-source ecosystems, and established security frameworks into actionable patterns for each vulnerability class — patterns designed to be enforced centrally, tested automatically, and executed at wire speed.

## Token-to-resource binding stops cross-resource reuse cold

The most insidious access control bug in collaborative sharing is a token issued for Board A that silently grants access to Board B. This is **IDOR (Insecure Direct Object Reference) applied to share tokens**, and OWASP ranks Broken Access Control as the #1 web application vulnerability, detected in 94% of tested applications.

Use opaque tokens with server-side lookup, not JWTs, for share links. An opaque token is a cryptographically random string that maps to a server-side record containing the `resource_id`, `permissions`, `expires_at`, and `is_revoked` fields. This design leaks no information to the client, supports instant revocation (delete the row), and makes the server the single source of truth. Store only the SHA-256 hash of the token — never plaintext. If JWTs must be used, embed `resource_id` and `resource_type` in the claims and still validate server-side that the claim matches the requested resource; the JWT is not self-sufficient for authorization.

The critical enforcement point is a single middleware that runs on **every request** involving a share token. The middleware must: (1) look up the token hash, (2) assert the token record's `resource_id` matches the resource ID in the request path, (3) check expiry and revocation, and (4) confirm the requested action falls within the token's permission scope. If any check fails, return 403 — and log the attempt as a potential attack. Figma implements per-file share links with configurable permissions and expiration windows from one hour to 31 days. Google Docs' internal authorization tokens embed `resource_name` and `role` claims, verified by Google's Key Access Control List Service before every operation.

Real-world breaches illustrate the pattern's importance. A Facebook Business Manager IDOR allowed attackers to create share links for campaign plans they didn't own, netting a **$5,375 bounty**. A FileBrowser CVE (GHSA-6cqf-cfhv-659g) let any authenticated user delete another user's share links because the handler checked the share hash without verifying ownership. In every case, the server validated that the token was structurally valid but failed to validate that the token was authorized for the specific resource.

## Fail-closed API contracts prevent silent access control breakdowns

When a client calls `/api/validate-share/{token}` and receives `{ valid: true }`, the entire access control decision pivots on that response. If the response is malformed, missing fields, or incorrectly returns `valid: true`, the client grants access it shouldn't.

The defense is **runtime schema validation on both sides of the contract**. On the server, initialize the response as `valid: false` and only set it to `true` after every check passes — the "default deny" pattern. On the client, validate the response against a strict schema using a library like Zod before acting on it. A `valid: true` response missing `resource_id` or `permissions` must be treated as access denied. This fail-closed principle ensures that API contract drift — a renamed field, a missing property, a type change — results in denial rather than accidental access.

Consumer-driven contract testing with Pact catches contract mismatches before they reach production. The frontend test declares exactly what shape it expects from the auth endpoint; the backend verifies it produces that shape. Property-based testing adds mathematical rigor: generate random resource ID pairs and assert that a token for resource A **never** validates for resource B. An authorization matrix test — enumerating every combination of token type, resource, and action — should run in CI on every commit. Figma's security team reviews features pre-launch and maintains reusable security libraries to prevent exactly this class of contract-level failure.

## Deny-by-default message dispatch is the only safe pattern for WebSocket mutations

This is where collaborative apps are most uniquely vulnerable. A whiteboard might have **50–100+ distinct WebSocket message types** — `object:move`, `object:resize`, `text:edit`, `paste`, `duplicate`, `undo`, `redo`, `style:change`, and more. Guarding each handler individually is a recipe for gaps. One forgotten handler means a read-only user can silently mutate shared state.

The proven architecture is a centralized message dispatcher with an explicit registry and deny-by-default semantics. Every message type is registered with a mandatory `requiresWrite` boolean — no default value, forcing developers to make a conscious permission decision for each new handler. Unknown message types are rejected outright. Read-only users are checked against a small **allowlist** of non-mutating operations (cursor position, viewport pan/zoom, presence heartbeat), and everything else is blocked. This is fundamentally safer than a blocklist because new features are blocked by default until explicitly approved.

Figma built an entire **custom permissions DSL (PermissionsV2)** after their original approach — calling `has_access?` in individual controllers — became unmanageable and produced bugs. The DSL centralizes all permission logic into a single, testable engine evaluated consistently across Ruby, TypeScript, and other services. Figma also built a **Response Sampling system**: middleware that samples API responses in staging, scans for resource identifiers, and retroactively verifies the requesting user had permission to each resource. This async safety net catches authorization flaws that static analysis misses.

ShareDB's `sharedb-access` plugin implements the deny-by-default model explicitly: "By default all operations are denied. You must add rules to allow them." The `submit` middleware fires for every OT operation with full context — the operation, the document, and the user session — making it impossible for a mutation to bypass the check. Hocuspocus (the production Yjs server) supports setting `connection.readOnly = true` during authentication, causing the server to drop all incoming Y.js updates from that connection at the CRDT sync layer.

For performance, **resolve permissions once at connection time and cache them on the socket object**. The per-message permission check becomes an in-memory Set lookup — O(1), microseconds — with zero database or network I/O in the hot path. Permission changes mid-session propagate via pub/sub, updating the cached state or forcing reconnection.

Testing must be automated and exhaustive. Enumerate all registered message types programmatically, then assert that every `requiresWrite: true` handler rejects viewer connections. A separate test should verify that every handler has an explicit `requiresWrite` declaration — any handler missing the declaration fails the build. OWASP recommends using ZAP to intercept, replay, and fuzz WebSocket messages, sending all known message types with viewer credentials and verifying rejection.

## Debug exposure is a CWE-489 vulnerability with a build-time fix

Debug query parameters (`?debug=true`, `?verbose=1`) that activate diagnostic panels, verbose logging, or internal state exposure in production are classified as **CWE-489: Active Debug Code**. MITRE notes these can create "unintended entry points" and at worst give attackers "complete control over the web application."

Build-time dead code elimination is the primary defense. Webpack's DefinePlugin replaces `process.env.NODE_ENV` with `'production'` at compile time, and Terser strips the resulting dead `if (false)` branches entirely. The critical caveat: the expression `process.env.NODE_ENV !== 'production'` must appear **inline** — abstracting it into a helper function defeats webpack's static analysis. React uses this pattern extensively; its production build includes a `checkDCE()` function that throws an error if dead code elimination hasn't been applied.

For debug tools that must exist in production for engineering support, gate them behind **feature flags with strict RBAC**. Treat the feature flag system as Tier-1 infrastructure: segregate tokens by environment, rotate backend tokens as secrets, restrict toggle permissions to authorized engineers, and audit every flag change. Runtime guards — server-side middleware that strips or rejects debug-related query parameters — add defense-in-depth but should never be the sole protection.

## UI hiding is user experience; server-side enforcement is security

Hiding editing tools, AI command bars, rename inputs, and context menu items from read-only users is a UX optimization, not a security boundary. As OWASP and every major collaborative platform emphasize, **any JavaScript running in the browser is fully readable and manipulable** — hiding a button does not prevent an attacker from calling the underlying API.

The scalable pattern is a **React Context-based Permission Provider** that wraps the application and feeds permission state to a `<Restricted>` component or `usePermission()` hook. Libraries like CASL provide an `ability.can('action', subject)` API that decouples UI rendering from role checks, making it trivial to add new roles without modifying component code. Bulk permission checks — a single API call returning which features a user can access — minimize round-trips when rendering complex dashboards.

The commonly forgotten UI elements are the dangerous ones: **keyboard shortcuts** (Ctrl+Z, Delete, Backspace), **right-click context menus** exposing Delete/Rename/Duplicate, **double-click-to-rename** behaviors on labels, **drag-and-drop** move/resize, **paste operations**, and **AI command interfaces** that can trigger mutations. Figma, Miro, and Google Docs all display a clear "View only" badge in the toolbar and strip editing tools entirely for viewers — but the server rejects mutations regardless of what the client sends.

For accessibility, use `aria-disabled="true"` rather than `display: none` when elements should be announced as unavailable, and provide contextual explanations ("Editing disabled — you have view-only access").

## Input sanitization in collaborative contexts extends far beyond XSS

Canvas rendering via `fillText()` is inherently safe from traditional XSS — it renders pixels, not DOM. But collaborative apps face a broader sanitization surface: **Unicode bidirectional override attacks, zero-width character abuse, Zalgo text, emoji floods, and SVG injection**.

The **Trojan Source attack** (CVE-2021-42574) uses Unicode bidirectional override characters (U+202A–U+202E, U+2066–U+2069) to reorder how text displays versus how it processes. In a shared document, malicious Bidi characters can mislead collaborators about content meaning. Zero-width characters (U+200B, U+200C, U+200D) create apparently blank usernames, enable impersonation of legitimate display names, and embed invisible fingerprinting data. Combining diacritical marks create "Zalgo text" that breaks layout boundaries.

The sanitization pipeline for display names, board names, cursor labels, and shape text should follow this order: **normalize first (NFC/NFKC), then validate, then sanitize at output time based on rendering context**. Strip zero-width and Bidi override characters from all user-facing labels. Enforce strict length limits (50 characters for display names, 100 for board names). Limit consecutive emoji, whitespace, and combining marks. For SVG content — critical in design tools — strip all `<script>` tags, event handlers, and `<foreignObject>` elements; serve user SVGs from a separate origin with `Content-Security-Policy` disallowing inline scripts.

CRDTs like Yjs and Automerge guarantee eventual consistency but not content validity. The CRDT layer merges operations mathematically; content validation — length limits, character filtering, schema enforcement — must be a **separate server-side layer** that intercepts operations before persistence and broadcast. Automerge explicitly delegates all authorization and validation to the application layer, while Hocuspocus can intercept Y.js updates at the server before relaying them.

## Conclusion

The six vulnerability classes covered here share a common architectural lesson: **centralize enforcement, deny by default, and never trust the client**. Token validation must bind to specific resources in a single middleware. API contracts must fail closed under schema violations. WebSocket mutation dispatch must route through one registry where unknown message types are rejected and permission declarations are mandatory. Debug code must be eliminated at build time, not toggled at runtime. UI hiding supplements but never replaces server enforcement. And input sanitization must account for Unicode abuse, not just HTML injection.

The performance concern is largely a false trade-off. Caching permissions on the socket object at connection time makes per-message authorization an in-memory lookup measured in microseconds. Figma's PermissionsV2 DSL was built specifically because their previous approach was both insecure *and* consuming 20% of database load — the centralized design was faster and safer. The most impactful testing investment is automated enumeration: programmatically walking all message types, all token-resource combinations, and all API response shapes to verify that the deny-by-default invariant holds everywhere, always.