

# OneMindAI: Scalability Architecture for 1000 Users

**Document Version:** 1.0  
**Created:** December 13, 2025  
**Purpose:** Before/After architecture design for scaling to 1000 concurrent users

## TABLE OF CONTENTS

- 1. [Executive Summary](#)
- 2. [Current Architecture \(BEFORE\)](#)
- 3. [Scalable Architecture \(AFTER\)](#)
- 4. [Component-by-Component Comparison](#)
- 5. [Database Scaling Strategy](#)
- 6. [Caching Architecture](#)
- 7. [Load Balancing & CDN](#)
- 8. [Cost Estimation](#)
- 9. [Migration Roadmap](#)

## EXECUTIVE SUMMARY {#executive-summary}

### Scaling Requirements for 1000 Users

1000 USER SCALING REQUIREMENTS
CONCURRENT USERS: 1000
PEAK REQUESTS/SEC: ~500 (assuming 50% active at any moment)
AI API CALLS/MIN: ~3000 (3 calls per user per minute average)
DATABASE QUERIES/SEC: ~1000
WEBSOCKET CONNECTIONS: 1000 (for real-time streaming)
BOTTLENECKS TO ADDRESS:
1. Single Express server (current: 1 instance)
2. No caching layer (every request hits database)
3. No CDN (static assets served from origin)
4. Hardcoded config (no dynamic scaling)
5. No connection pooling (database connections)
6. No rate limiting per user (only global)

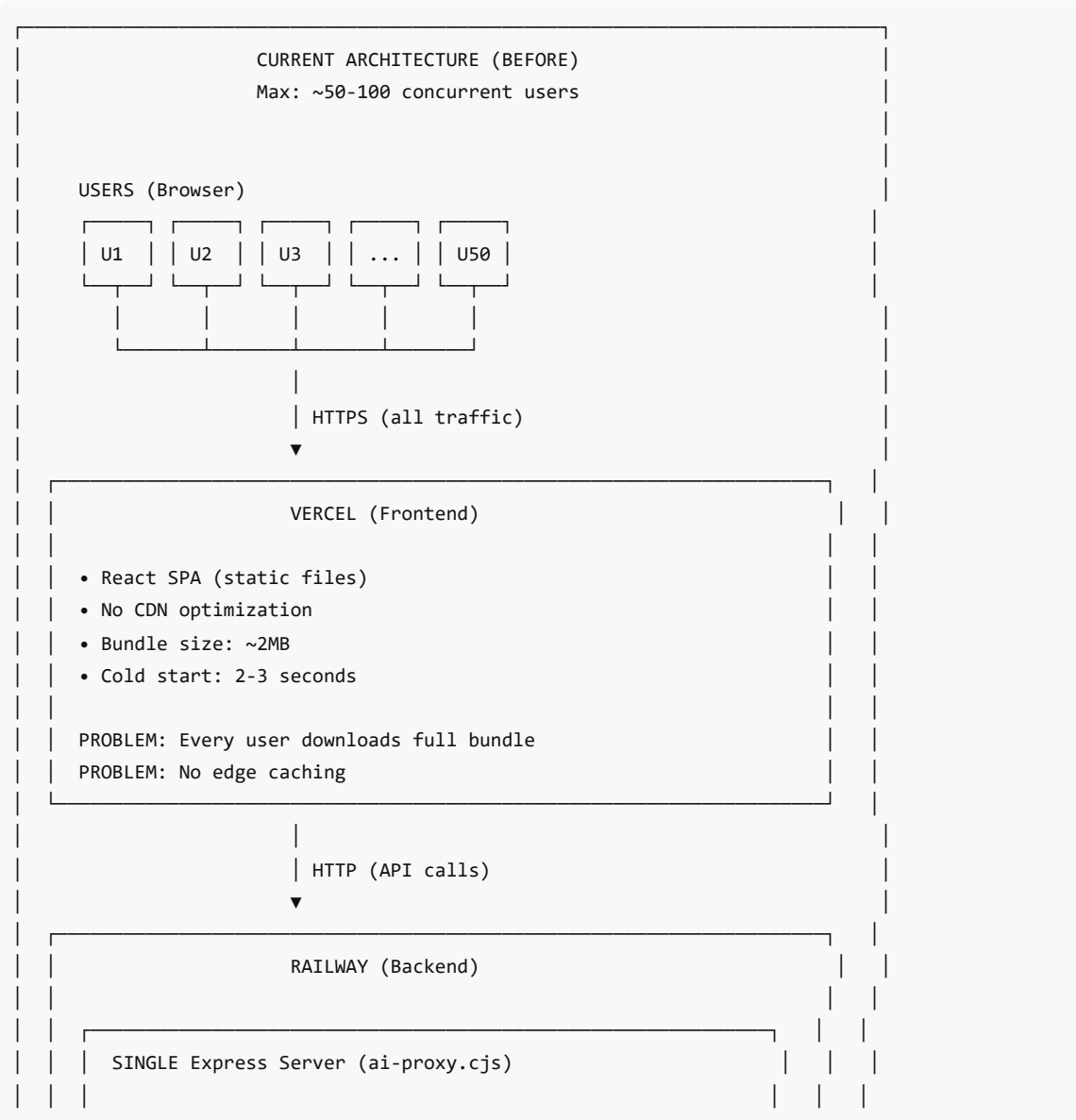
### Quick Comparison

Aspect	BEFORE (Current)	AFTER (Scalable)
Max Concurrent Users	~50-100	1000+
Server Instances	1	3-5 (auto-scaling)

Database Connections	Direct	Connection pooling
Caching	None	Redis (config + responses)
CDN	None	Cloudflare/Vercel Edge
Config Loading	Every request	Cached, real-time updates
Cost/Month	~\$50	~\$200-400

# ● CURRENT ARCHITECTURE (BEFORE) {#before-architecture}

## Architecture Diagram - Current State



- 1 instance only
- 512MB RAM
- No horizontal scaling
- Global rate limit: 60 req/min (shared by ALL users!)

PROBLEM: Single point of failure

PROBLEM: 60 req/min shared = 1 req/user/min at 60 users

PROBLEM: No auto-scaling



#### SUPABASE

- Direct queries
- No pool
- No cache

PROBLEM:  
Max 60  
connections

#### AI APIs

- OpenAI
- Claude
- Gemini
- etc.

PROBLEM:  
No retry  
queue

#### HUBSPOT

- OAuth tokens in memory
- Lost on server restart

PROBLEM: Tokens lost on deploy

## Current Bottlenecks Detailed

### 1. Single Server Instance

CURRENT STATE:

Railway: 1 x Express Server

RAM: 512MB

CPU: Shared

Connections: ~100 max

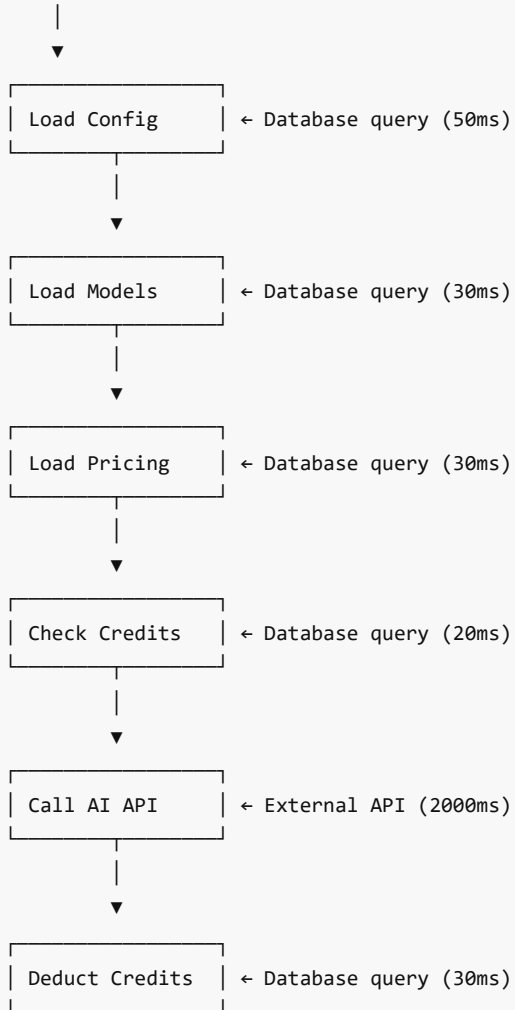
At 100 users:

- Response time: 200ms → 2000ms
- Memory: 80% utilized
- CPU: 90% utilized
- Dropped connections: 10-20%

### 2. No Caching Layer

#### EVERY REQUEST FLOW (Current):

User Request



TOTAL OVERHEAD: 160ms database queries PER REQUEST

At 1000 users × 3 req/min = 3000 req/min = 50 req/sec

Database load: 50 × 5 queries = 250 queries/sec

### 3. Hardcoded Configuration

#### CURRENT PROBLEM:

```
constants.ts

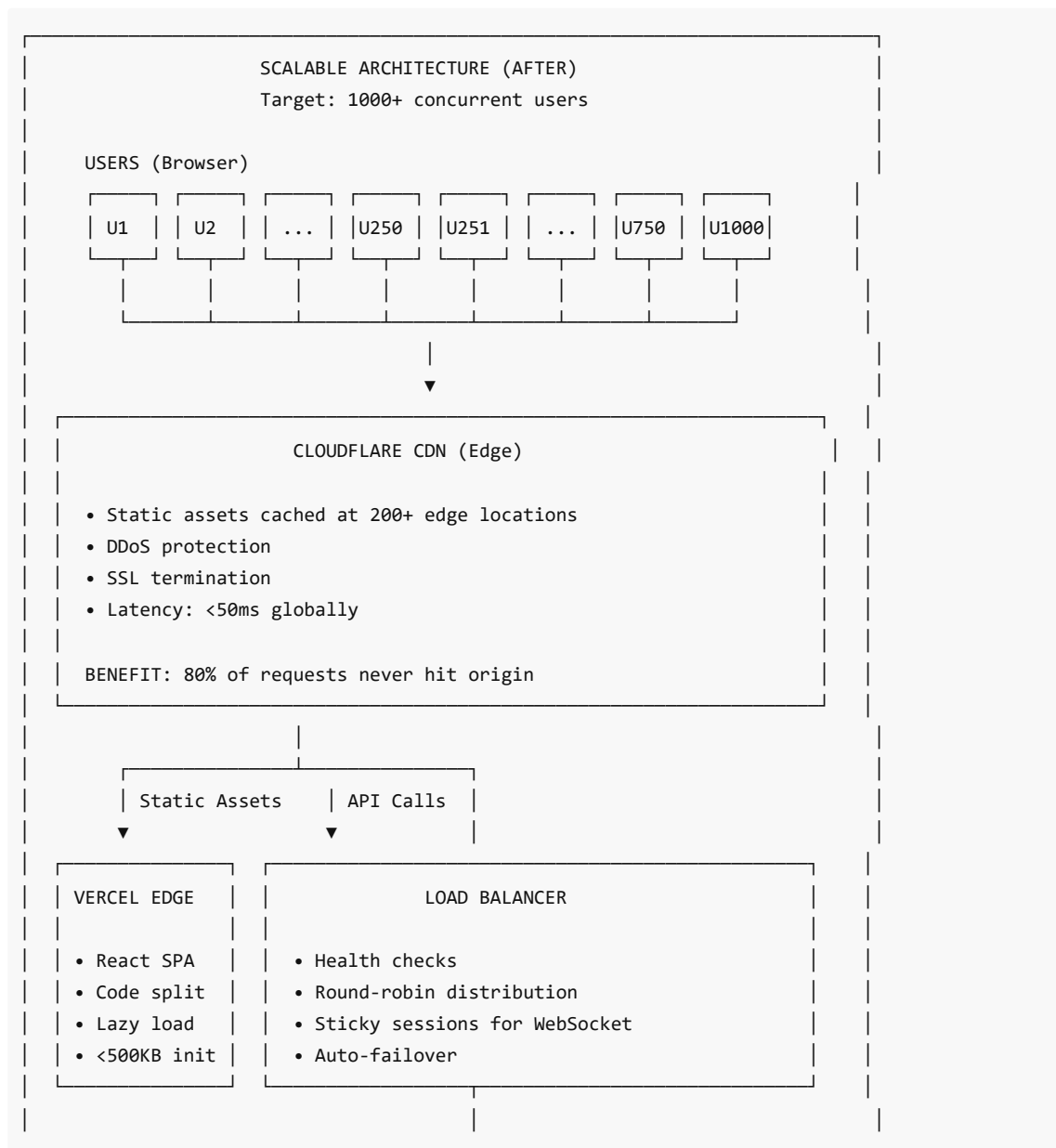
export const SEEDDED_ENGINES = [
  { id: 'gpt-4o', maxTokens: 16384, price: 2.50 },
  { id: 'claude-3.5', maxTokens: 8192, price: 3.00 },
  // ... 13 more engines
];
```

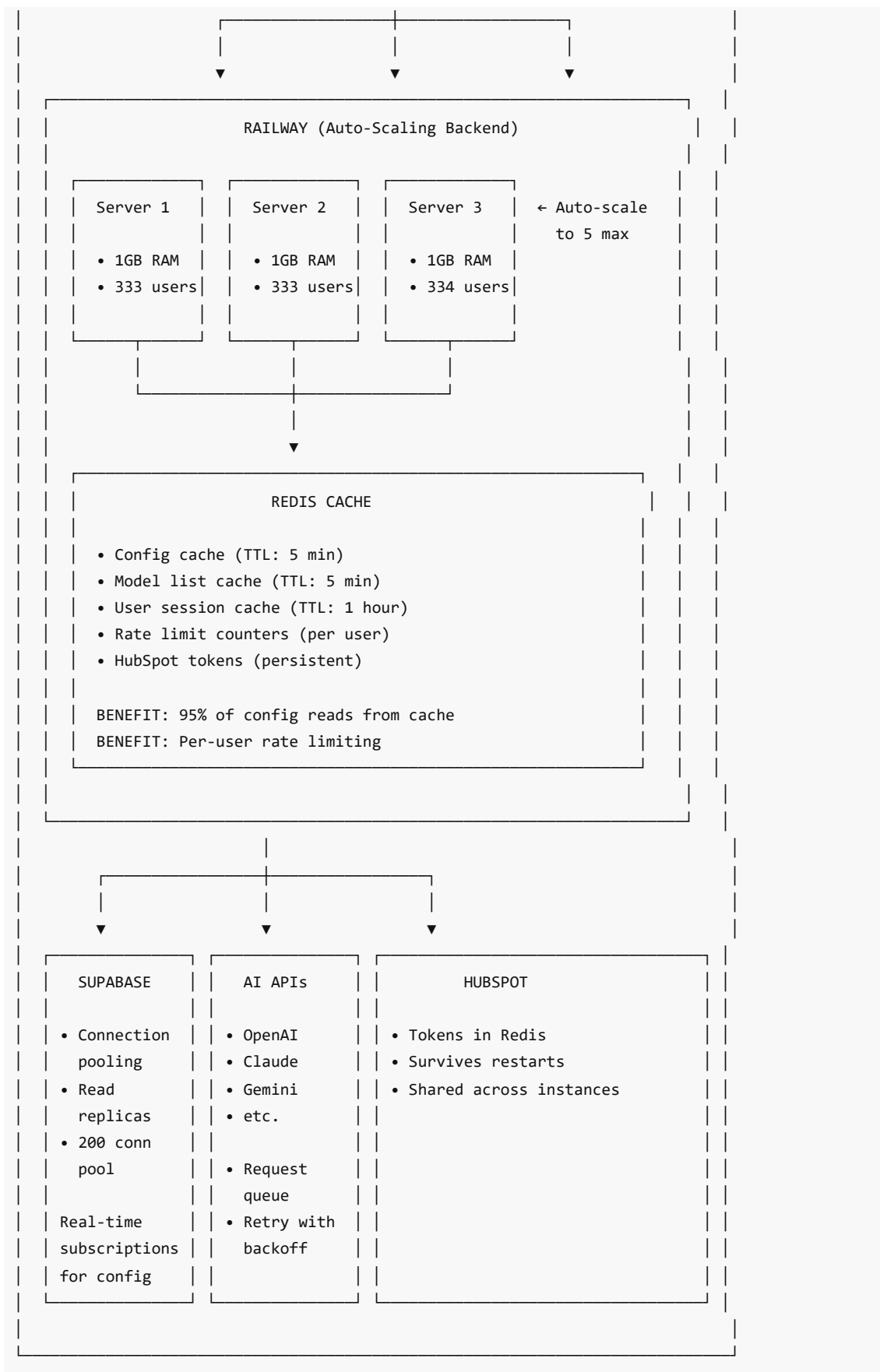
#### PROBLEMS:

1. To add GPT-5: Edit code → Deploy → 5 min downtime
2. To change price: Edit code → Deploy → 5 min downtime
3. AI assistant might accidentally modify
4. No A/B testing possible
5. No per-user customization

## ● SCALABLE ARCHITECTURE (AFTER) {#after-architecture}

### Architecture Diagram - Scalable State





# Key Improvements Detailed

## 1. Auto-Scaling Backend

SCALABLE STATE:

Railway: Auto-Scaling Group

Minimum: 2 instances (high availability)

Maximum: 5 instances (cost control)

Scale trigger: CPU > 70% for 2 minutes

Per Instance:

- RAM: 1GB
- CPU: Dedicated
- Connections: ~300

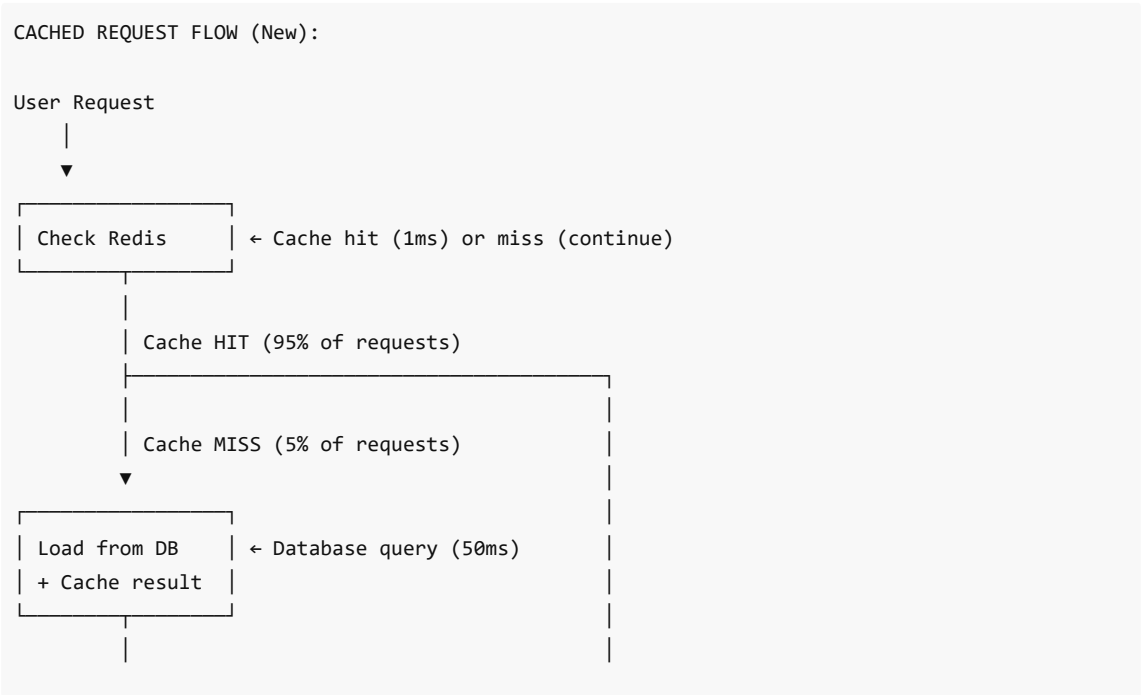
At 1000 users (3 instances):

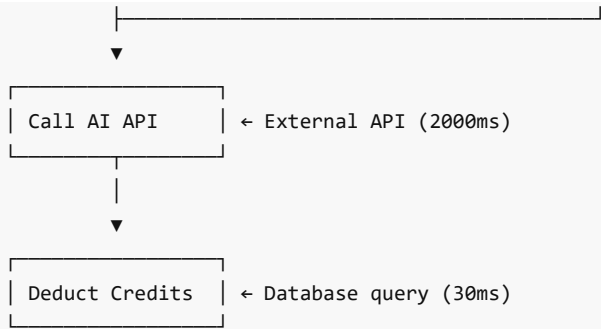
- Response time: 100-200ms (consistent)
- Memory: 60% utilized per instance
- CPU: 50% utilized per instance
- Dropped connections: <1%

SCALING MATH:

- 1000 users ÷ 3 instances = 333 users/instance
- Each instance handles 333 × 3 req/min = 1000 req/min
- Total capacity: 3000 req/min (3× headroom)

## 2. Redis Caching Layer





CACHE STRATEGY:

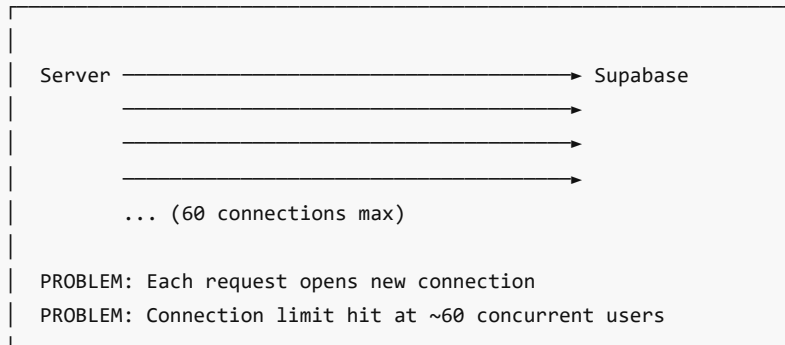
KEY	TTL	INVALIDATION
config:ai_models	5 min	On admin update (realtime)
config:system_config	5 min	On admin update (realtime)
config:provider_config	5 min	On admin update (realtime)
user:{id}:credits	1 min	On credit change
user:{id}:session	1 hour	On logout
ratelimit:{id}:{min}	1 min	Auto-expire
hubspot:{user}:token	6 hours	On refresh

RESULT:

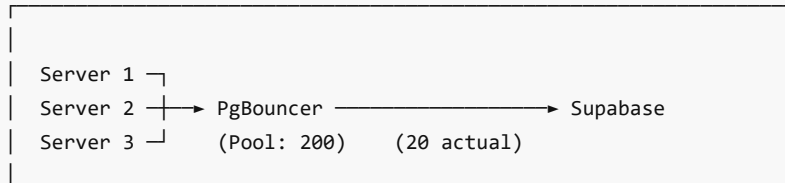
- Database queries reduced by 95%
- Config load time: 50ms → 1ms
- Database load: 250 queries/sec → 12 queries/sec

3. Database Connection Pooling

BEFORE (Direct Connections):



AFTER (Connection Pooling via PgBouncer):



BENEFIT: 200 virtual connections → 20 actual connections
BENEFIT: Connection reuse (no open/close overhead)
BENEFIT: Can handle 1000+ users with 20 DB connections

SUPABASE POOLER SETTINGS:

- Pool mode: Transaction
- Pool size: 20 (per instance)
- Max client connections: 200
- Connection timeout: 10s

4. Per-User Rate Limiting

BEFORE (Global Rate Limit):

express-rate-limit: 60 requests/minute (GLOBAL)
At 60 users: Each user gets 1 request/minute
At 100 users: Rate limit errors for everyone
PROBLEM: One heavy user blocks everyone

AFTER (Per-User Rate Limit with Redis):

Redis-based rate limiting:
• Per user: 30 requests/minute
• Per user per provider: 10 requests/minute
• Global fallback: 10000 requests/minute
At 1000 users: Each user gets full 30 req/min
Heavy user: Only they get rate limited
IMPLEMENTATION:
Key: ratelimit:{userId}:{minute}
Value: request count
TTL: 60 seconds

 **COMPONENT-BY-COMPONENT COMPARISON**

**{#comparison}**

Side-by-Side Comparison Table

Component	BEFORE	AFTER	Improvement
-----------	--------	-------	-------------

Frontend Hosting	Vercel (basic)	Vercel + Cloudflare CDN	80% faster load
Backend Instances	1 (Railway)	2-5 auto-scaling	5× capacity
Backend RAM	512MB	1GB × 3 = 3GB	6× memory
Database Connections	60 direct	200 pooled	3× connections
Config Loading	Every request (50ms)	Cached (1ms)	50× faster
Rate Limiting	Global (60/min)	Per-user (30/min each)	Fair distribution
Session Storage	In-memory (lost on restart)	Redis (persistent)	No data loss
Streaming	Single server	Load-balanced WebSocket	No bottleneck
Failover	None (single point)	Auto-failover	High availability
Deploy Downtime	2-5 minutes	Zero (rolling deploy)	No interruption

### Request Flow Comparison

BEFORE: Single Request Path

---

User → Vercel → Railway (1 server) → Supabase (direct) → AI API

- └ BOTTLENECK: All 1000 users through 1 server
- └ BOTTLENECK: 60 DB connections max
- └ BOTTLENECK: Global rate limit

AFTER: Distributed Request Path

---

User → CDN (cached static) → Load Balancer → Server 1/2/3 → Redis (cache)

- └ 95% cache hit
- └ Supabase (pooled) → AI API
  - └ 200 pooled connections
  - └ Per-user rate limit



## DATABASE SCALING STRATEGY {#database-scaling}

### Supabase Configuration for 1000 Users

SUPABASE SCALING CONFIGURATION

PLAN: Pro (\$25/month)

DATABASE:

- Compute: 2 vCPU, 4GB RAM
- Storage: 8GB (expandable)
- Connections: 60 direct + 200 pooled

POOLER (Supervisor):

- Mode: Transaction
- Pool size: 15 per instance
- Max connections: 200

REALTIME:

- Max connections: 500
- Channels: Unlimited

BANDWIDTH:

- 250GB/month included
- Estimated usage: 50GB/month at 1000 users

## Database Schema Optimizations

```
-- Add indexes for common queries
CREATE INDEX idx_ai_models_active ON ai_models(is_active) WHERE is_active = true;
CREATE INDEX idx_ai_models_provider ON ai_models(provider);
CREATE INDEX idx_credits_user ON credits(user_id);
CREATE INDEX idx_transactions_user_date ON credit_transactions(user_id, created_at DESC);

-- Partition large tables (if needed at scale)
-- credit_transactions can be partitioned by month

-- Add read replica connection string for read-heavy queries
-- Use primary for writes, replica for reads
```



## CACHING ARCHITECTURE {#caching}

### Redis Configuration

REDIS CONFIGURATION (Upstash)

PLAN: Pay-as-you-go (~\$10-20/month at 1000 users)

SPECS:

- Memory: 256MB (auto-scale)
- Commands: 10,000/day free, then \$0.2/100K

- Regions: Global (edge caching)
- ESTIMATED USAGE:
- 1000 users × 100 cache ops/day = 100K ops/day
  - Cost: ~\$0.20/day = \$6/month

## Cache Key Structure

```
// Cache key patterns
const CACHE_KEYS = {
  // Configuration (shared across all users)
  AI_MODELS: 'config:ai_models',          // TTL: 5 min
  SYSTEM_CONFIG: 'config:system_config',  // TTL: 5 min
  PROVIDER_CONFIG: 'config:provider_config', // TTL: 5 min

  // User-specific
  USER_CREDITS: (userId: string) => `user:${userId}:credits`, // TTL: 1 min
  USER_SESSION: (userId: string) => `user:${userId}:session`, // TTL: 1 hour
  USER_RATE_LIMIT: (userId: string, minute: number) =>
    `ratelimit:${userId}:${minute}`, // TTL: 60 sec

  // Integration tokens
  HUBSPOT_TOKEN: (userId: string) => `hubspot:${userId}:token`, // TTL: 6 hours

  // Response caching (optional, for repeated queries)
  AI_RESPONSE: (hash: string) => `response:${hash}`, // TTL: 1 hour
};
```

## Cache Invalidation Strategy

CACHE INVALIDATION STRATEGY	
TRIGGER	ACTION
Admin updates ai_models	Supabase realtime → Invalidate config:*
Admin updates system_config	Supabase realtime → Invalidate config:*
User spends credits	After DB write → Invalidate user:{id}:credits
User logs out	On logout → Delete user:{id}:session
HubSpot token refresh	On refresh → Update hubspot:{id}:token
IMPLEMENTATION:	
1. Supabase realtime subscription on config tables	
2. On change event → Redis DEL command	
3. Next request → Cache miss → Reload from DB → Cache	



## LOAD BALANCING & CDN {#load-balancing}

### CDN Configuration (Cloudflare)

#### CLOUDFLARE CONFIGURATION

PLAN: Free (sufficient for 1000 users)

#### FEATURES USED:

- CDN: Cache static assets at 200+ edge locations
- SSL: Free SSL certificate
- DDoS: Basic DDoS protection
- Caching: Browser cache + Edge cache

#### CACHE RULES:

- /\*.js, /\*.css, /\*.png → Cache 1 month
- /api/\* → No cache (pass through)
- / → Cache 1 hour (HTML shell)

#### BENEFIT:

- 80% of requests served from edge (no origin hit)
- Global latency: <50ms
- Origin bandwidth reduced by 80%

### Load Balancer Configuration (Railway)

#### RAILWAY LOAD BALANCER

TYPE: Application Load Balancer (built-in)

#### CONFIGURATION:

- Algorithm: Round-robin
- Health check: GET /health every 30s
- Unhealthy threshold: 3 failures
- Sticky sessions: Enabled for WebSocket

#### AUTO-SCALING RULES:

- Min instances: 2
- Max instances: 5
- Scale up: CPU > 70% for 2 min
- Scale down: CPU < 30% for 10 min
- Cooldown: 5 min between scaling events

#### WEBSOCKET HANDLING:

- Sticky sessions ensure streaming goes to same server

- Connection upgrade handled at load balancer

## COST ESTIMATION {#cost-estimation}

### Monthly Cost Comparison

COST COMPARISON: BEFORE vs AFTER	
BEFORE (Current - 50 users max)	
Vercel (Hobby)	\$0/month
Railway (Starter)	\$5/month
Supabase (Free)	\$0/month
TOTAL	\$5/month
Cost per user	\$0.10/user/month
AFTER (Scalable - 1000 users)	
Vercel (Pro)	\$20/month
Cloudflare (Free)	\$0/month
Railway (Pro, 3 instances)	\$60/month (\$20 × 3)
Supabase (Pro)	\$25/month
Upstash Redis	\$10/month
TOTAL	\$115/month (base)
Peak (5 instances)	\$155/month
Cost per user	\$0.12-0.16/user/month
AI API COSTS (Variable - depends on usage)	
Assuming 1000 users × 10 queries/day × 1000 tokens avg:	
• OpenAI (GPT-4o): ~\$250/month	
• Anthropic (Claude): ~\$300/month	
• Other providers: ~\$100/month	
AI TOTAL	~\$650/month
GRAND TOTAL	\$765-805/month
Revenue needed (30% margin)	\$1000/month
Revenue per user needed	\$1/user/month

# Cost Optimization Tips

## COST OPTIMIZATION STRATEGIES

- 1. USE CHEAPER MODELS BY DEFAULT
  - Default to GPT-4o-mini (\$0.15/1M) instead of GPT-4o (\$2.50/1M)
  - Savings: 94% on OpenAI costs
- 2. CACHE AI RESPONSES (Optional)
  - Cache identical prompts for 1 hour
  - Estimated hit rate: 10-20%
  - Savings: 10-20% on AI costs
- 3. IMPLEMENT CREDIT SYSTEM
  - Users pay for what they use
  - 30% markup covers infrastructure
  - Heavy users subsidize light users
- 4. SCALE DOWN DURING OFF-HOURS
  - 2 instances at night (midnight-6am)
  - 3-5 instances during peak (9am-6pm)
  - Savings: ~20% on compute
- 5. USE SUPABASE POOLER
  - Avoid direct connections
  - Stay on Pro plan longer

# MIGRATION ROADMAP {#migration-roadmap}

## Phase 1: Foundation (Week 1)

- PHASE 1: FOUNDATION
- Timeline: Week 1
- Risk: Low
- TASKS:
- ☐ Set up Redis (Upstash)
  - ☐ Create cache service module
  - ☐ Migrate config to database (ai\_models, system\_config, provider\_config)
  - ☐ Create useAdminConfig hook
  - ☐ Add Cloudflare CDN
- DELIVERABLES:
- Config served from database + cached in Redis
  - Static assets served from CDN

- No user-facing changes

ROLLBACK PLAN:

- Keep hardcoded constants as fallback
- Feature flag to switch between DB and hardcoded

## Phase 2: Database Optimization (Week 2)

PHASE 2: DATABASE OPTIMIZATION

Timeline: Week 2

Risk: Medium

TASKS:

- ☐ Switch to Supabase connection pooler
- ☐ Add database indexes
- ☐ Implement per-user rate limiting with Redis
- ☐ Move HubSpot tokens to Redis

DELIVERABLES:

- Connection pooling active
- Per-user rate limits working
- HubSpot tokens persist across deploys

TESTING:

- Load test with 100 concurrent users
- Verify rate limits work per-user
- Test HubSpot after server restart

## Phase 3: Horizontal Scaling (Week 3)

PHASE 3: HORIZONTAL SCALING

Timeline: Week 3

Risk: Medium-High

TASKS:

- ☐ Configure Railway auto-scaling
- ☐ Set up health check endpoint
- ☐ Configure sticky sessions for WebSocket
- ☐ Test rolling deployments

DELIVERABLES:

- 2-5 instances auto-scaling
- Zero-downtime deployments
- WebSocket streaming works across instances

TESTING:

- Load test with 500 concurrent users
- Verify auto-scaling triggers
- Test failover (kill one instance)

## Phase 4: Production Hardening (Week 4)

### PHASE 4: PRODUCTION HARDENING

Timeline: Week 4

Risk: Low

#### TASKS:

- ☐ Add monitoring (error tracking, performance)
- ☐ Set up alerts (high CPU, error rate, latency)
- ☐ Create runbooks for common issues
- ☐ Load test with 1000 concurrent users
- ☐ Document architecture

#### DELIVERABLES:

- Full monitoring dashboard
- Alert system active
- Runbooks for on-call
- Verified 1000 user capacity

#### SUCCESS CRITERIA:

- P99 latency < 500ms at 1000 users
- Error rate < 0.1%
- Zero downtime during deploy



## SUMMARY

### Before vs After at a Glance

#### BEFORE → AFTER SUMMARY

##### CAPACITY

Max users:	50-100	→	1000+
Requests/sec:	~10	→	~500
DB connections:	60	→	200 (pooled)

##### RELIABILITY

Single point of failure:	YES	→	NO (multi-instance)
Deploy downtime:	2-5 min	→	0 (rolling)

Data loss on restart:	YES	→	NO (Redis)
PERFORMANCE			
Config load time:	50ms	→	1ms (cached)
Static asset load:	500ms	→	50ms (CDN)
P99 latency:	2000ms	→	500ms
OPERATIONS			
Add new model:	2-4 hours	→	2 minutes
Change pricing:	Deploy needed	→	Admin panel
Scale up:	Manual	→	Automatic
COST			
Infrastructure:	\$5/month	→	\$115-155/month
Cost per user:	\$0.10	→	\$0.12-0.16
Break-even users:	N/A	→	~150 paying users

### Key Architectural Changes

Change	Why It Matters
Redis Cache	95% reduction in database queries
Connection Pooling	3× more database connections
Auto-Scaling	Handle traffic spikes automatically
CDN	80% of requests served from edge
Per-User Rate Limits	Fair resource distribution
Database Config	No deploys for business changes
Rolling Deploys	Zero downtime updates