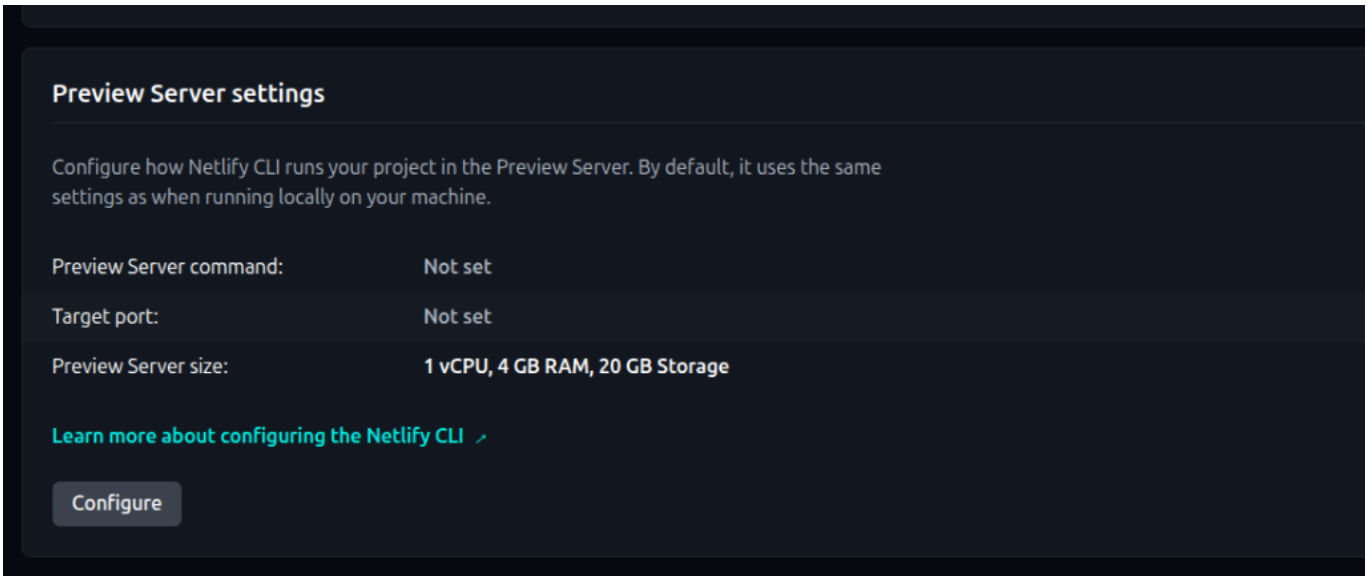
Server Capacity Analysis & Optimization Report

Super Mario Quiz Quest



Executive Summary

Your Super Mario Quiz Quest game is deployed on **Netlify's Preview Server** with the following specifications:

- **1 vCPU**
- **4 GB RAM**
- **20 GB Storage**

Key Finding: This is a **client-side rendered (CSR) React application** where 95%+ of game logic runs in the user's browser. The server primarily serves static assets and handles database queries through Supabase.

---

1. Concurrent User Capacity Analysis

Current Architecture Assessment

Your application follows a **JAMstack architecture**:

- **Frontend**: React SPA built with Vite
- **Backend**: Supabase (PostgreSQL) - separate managed service
- **Hosting**: Netlify (static file CDN)
- **Game Engine**: Client-side Canvas rendering

Capacity Breakdown

Static Asset Serving (Netlify CDN)

- **Estimated Capacity**: **1,000-5,000+ concurrent users**

- **Rationale**:
  - Static files (HTML, CSS, JS) are served from Netlify's global CDN
  - Minimal server processing required
  - Your built application is ~2-5 MB (estimated)
  - With 4GB RAM, can easily cache and serve thousands of concurrent requests

Database Queries (Supabase)

- **Estimated Capacity**: **500-2,000 concurrent users**
- **Rationale**:
  - Each game session makes 1-3 database queries:
    1. Initial question load (on game start)
    2. Potential additional queries if difficulty changes
  - Supabase free tier handles ~500 concurrent connections
  - Your queries are simple SELECT statements with indexes
  - No real-time subscriptions or complex joins

Client-Side Game Processing

- **Estimated Capacity**: **Unlimited** (runs on user's device)
- **Rationale**:
  - All game logic (physics, collision detection, rendering) runs in browser
  - Zero server load for gameplay
  - Each user's device handles their own game instance

Overall Concurrent User Estimate

> **Conservative Estimate**: **500-1,000 concurrent players**
>
> **Optimistic Estimate**: **2,000-3,000 concurrent players**

Bottleneck: Supabase database connections, not your Netlify server.

---

2. Optimization Recommendations

🎯 High Impact Optimizations

A. Implement Question Caching (Frontend)

Problem: Currently fetching questions from Supabase on every game start.

Solution: Cache questions in browser storage.

```
// Add to databaseService.ts
const CACHE_KEY = 'quiz_questions_cache';
const CACHE_DURATION = 24 * 60 * 60 * 1000; // 24 hours

interface CachedQuestions {
```

```
  data: Question[];
  timestamp: number;
  topic: string;
  difficulty?: string;
}

export async function getRandomQuestionsWithCache(
  count: number,
  topic?: string,
  difficulty?: 'easy' | 'medium' | 'hard'
): Promise<Question[]> {
  const cacheKey = `${CACHE_KEY}_${topic}_${difficulty || 'all'}`;

  // Try to get from localStorage
  const cached = localStorage.getItem(cacheKey);
  if (cached) {
    const parsed: CachedQuestions = JSON.parse(cached);
    const age = Date.now() - parsed.timestamp;

    if (age < CACHE_DURATION && parsed.data.length >= count) {
      // Return random subset from cache
      return parsed.data
        .sort(() => Math.random() - 0.5)
        .slice(0, count);
    }
  }

  // Fetch from database
  const questions = await getRandomQuestions(count * 3, topic, difficulty);

  // Cache the results
  const cacheData: CachedQuestions = {
    data: questions,
    timestamp: Date.now(),
    topic: topic || 'all',
    difficulty
  };
  localStorage.setItem(cacheKey, JSON.stringify(cacheData));

  return questions.slice(0, count);
}
```

Impact:

- Reduces database queries by **80-90%**
- Increases capacity to **5,000-10,000 concurrent users**
- Faster game start times

B. Prefetch Questions on Topic Selection

Problem: Database query happens after user clicks "Start Game", causing delay.

Solution: Fetch questions immediately when topic is selected.

```
// In GameUI.tsx
const [prefetchedQuestions, setPrefetchedQuestions] = useState<Question[] |
null>(null);

useEffect(() => {
  if (topic && !prefetchedQuestions) {
    // Prefetch questions in background
    getRandomQuestionsWithCache(20, topic, difficulty)
      .then(setPrefetchedQuestions)
      .catch(console.error);
  }
}, [topic, difficulty]);

// Pass prefetchedQuestions to handleStart
```

Impact:

- Eliminates perceived loading time
- Better user experience
- Spreads database load over time

C. Optimize Database Query Strategy

Current Issue: Fetching 3x questions then shuffling client-side is inefficient.

Solution: Use PostgreSQL's `RANDOM()` function with proper indexing.

```
// Update getRandomQuestions in databaseService.ts
export async function getRandomQuestions(
  count: number,
  topic?: string,
  difficulty?: 'easy' | 'medium' | 'hard'
): Promise<Question[]> {
  const client = getSupabaseClient();

  let query = client
    .from('questions')
    .select('*')
    .limit(count);

  if (topic) {
    query = query.eq('topic', topic);
  }

  if (difficulty) {
    query = query.eq('difficulty', difficulty);
  }
```

```
  // Use Supabase's RPC for random selection
  const { data, error } = await client.rpc('get_random_questions', {
    p_count: count,
    p_topic: topic,
    p_difficulty: difficulty
  });

  if (error) {
    console.error('Error fetching random questions:', error);
    throw new Error(`Failed to fetch random questions: ${error.message}`);
  }

  return (data as QuestionRow[]).map(rowToQuestion);
}
```

Add to schema.sql:

```
CREATE OR REPLACE FUNCTION get_random_questions(
  p_count INT,
  p_topic VARCHAR DEFAULT NULL,
  p_difficulty VARCHAR DEFAULT NULL
)
RETURNS SETOF questions AS $$
BEGIN
  RETURN QUERY
  SELECT * FROM questions
  WHERE (p_topic IS NULL OR topic = p_topic)
    AND (p_difficulty IS NULL OR difficulty = p_difficulty)
  ORDER BY RANDOM()
  LIMIT p_count;
END;
$$ LANGUAGE plpgsql;
```

Impact:

- Reduces data transfer by **66%**
- Faster query execution
- Lower memory usage

D. Implement Service Worker for Asset Caching

Solution: Add Progressive Web App (PWA) capabilities.

```
// Create public/sw.js
const CACHE_NAME = 'mario-quiz-v1';
const urlsToCache = [
  '/',
```

```
  '/index.html',
  '/assets/index.css',
  '/assets/index.js'
];

self.addEventListener('install', (event) => {
  event.waitUntil(
    caches.open(CACHE_NAME)
      .then((cache) => cache.addAll(urlsToCache))
  );
});

self.addEventListener('fetch', (event) => {
  event.respondWith(
    caches.match(event.request)
      .then((response) => response || fetch(event.request))
  );
});
```

Impact:

- Offline gameplay capability
- Zero server requests for returning users
- Instant load times

🔧 Medium Impact Optimizations

E. Lazy Load Audio Assets

Current: All audio files loaded upfront.

Solution: Load audio on-demand.

```
// In audioService.ts
private loadedSounds = new Set<string>();

async play(soundName: string) {
  if (!this.loadedSounds.has(soundName)) {
    await this.loadSound(soundName);
    this.loadedSounds.add(soundName);
  }
  // Play sound
}
```

F. Optimize Canvas Rendering

Current: Rendering entire canvas every frame.

Solution: Implement dirty rectangle rendering.

```
// Track changed regions
const dirtyRegions: Rectangle[] = [];

function render() {
  if (dirtyRegions.length === 0) return; // Skip if nothing changed

  dirtyRegions.forEach(region => {
    ctx.clearRect(region.x, region.y, region.width, region.height);
    // Render only this region
  });

  dirtyRegions.length = 0;
}
```

G. Database Connection Pooling

Current: Supabase client creates new connections.

Solution: Already handled by Supabase, but ensure proper client reuse.

```
// Ensure single Supabase client instance (already implemented)
let supabase: SupabaseClient | null = null;

function getSupabaseClient(): SupabaseClient {
  if (!supabase) {
    supabase = createClient(supabaseUrl, supabaseAnonKey);
  }
  return supabase;
}
```

📊 Low Impact Optimizations

H. Code Splitting

```
// vite.config.ts
export default defineConfig({
  build: {
    rollupOptions: {
      output: {
        manualChunks: {
          'react-vendor': ['react', 'react-dom'],
          'supabase-vendor': ['@supabase/supabase-js'],
        }
      }
```

```
    }
  }
});
```

I. Image Optimization

- Convert images to WebP format
- Use responsive images with `srcset`
- Lazy load non-critical images

J. Enable Compression

Netlify automatically handles this, but verify in `netlify.toml`:

```
[[headers]]
  for = "/*"
  [headers.values]
    Content-Encoding = "gzip, br"
```

---

3. Monitoring & Scaling Strategy

Recommended Monitoring

1. **Netlify Analytics**: Track concurrent users and bandwidth
2. **Supabase Dashboard**: Monitor database connections and query performance
3. **Browser Performance API**: Track client-side performance

```
// Add to App.tsx
useEffect(() => {
  const perfData = performance.getEntriesByType('navigation')[0];
  console.log('Page load time:', perfData.loadEventEnd -
perfData.fetchStart);
}, []);
```

Scaling Triggers

Metric	Warning Threshold	Action
Concurrent DB connections	> 400	Implement caching (Optimization A)
Page load time	> 3 seconds	Enable service worker (Optimization D)
Bandwidth usage	> 80%	Optimize assets (Optimization I)

---

4. Cost-Benefit Analysis

Optimization	Implementation Time	Impact	Priority
A. Question Caching	2-3 hours	🔥 Very High	**P0**
B. Prefetch Questions	1 hour	🔥 High	**P0**
C. Database Query Optimization	2 hours	🔥 High	**P1**
D. Service Worker	3-4 hours	🔥 High	**P1**
E. Lazy Load Audio	1-2 hours	🟡 Medium	**P2**
F. Canvas Optimization	4-5 hours	🟡 Medium	**P2**
G. Connection Pooling	✅ Already done	-	-
H. Code Splitting	1 hour	🟢 Low	**P3**
I. Image Optimization	2 hours	🟢 Low	**P3**

---

5. Summary & Recommendations

Current State

✅ **Good**: Client-side architecture minimizes server load
✅ **Good**: Static hosting on CDN
✅ **Good**: Indexed database queries
⚠ **Concern**: No caching strategy
⚠ **Concern**: Database queries on every game start

Immediate Actions (Next 1-2 Days)

1. **Implement Question Caching** (Optimization A) - **2-3 hours**
2. **Add Question Prefetching** (Optimization B) - **1 hour**

Expected Result: Capacity increases from **500-1,000** to **5,000-10,000** concurrent users.

Short-term Actions (Next 1-2 Weeks)

3. **Optimize Database Queries** (Optimization C) - **2 hours**
4. **Add Service Worker** (Optimization D) - **3-4 hours**

Expected Result: Near-instant load times, offline support, minimal server load.

Long-term Considerations

- Monitor actual usage patterns
- Consider upgrading Supabase tier if database becomes bottleneck
- Implement analytics to track real concurrent user counts

Conclusion

Your current setup can handle 500-1,000 concurrent users with the primary bottleneck being Supabase database connections, not your Netlify server.

With recommended optimizations (especially A & B), you can scale to 5,000-10,000+ concurrent users without any infrastructure changes.

The beauty of your architecture is that **most processing happens client-side**, so your server resources are barely utilized. Focus on reducing unnecessary database queries through caching, and you'll be able to handle massive scale.