Architektur, Skalierungsplan & Fahrplan für Next.js Cybersicherheits-Scanner

Dieses Dokument beschreibt nicht nur die Architektur und Skalierung, sondern gibt dir auch einen klaren Schritt-für-Schritt-Plan, um von 0 auf 1.000.000 Nutzer zu wachsen.

1. Grundaufbau für 100.000 Nutzer

Für 100.000 Nutzer reicht eine einfachere Architektur: - Next.js als Frontend & API für Status-Abfragen - Separater Node.js-Service für die eigentliche Scanner-Logik - Eine zentrale Datenbank (z. B. PostgreSQL) - Redis als Cache für schnelle Statusupdates - Job-Queue (BullMQ oder RabbitMQ) für asynchrone Verarbeitung der Scans - Hosting: Vercel/Netlify für Frontend, AWS/GCP/DigitalOcean für Backend

Komponente	Technologie	Funktion
Frontend	Next.js	UI, Statusabfragen
Backend	Node.js (Express/NestJS)	Scanner-Logik, API für Scans
Datenbank	PostgreSQL/MongoDB	Speichert Nutzer & Ergebnisse
Queue	BullMQ/RabbitMQ	Verwaltet Scan-Jobs
Cache	Redis	Schnelle Statusupdates

2. Skalierung auf 1.000.000 Nutzer

Ab 1.000.000 Nutzern ist eine horizontale Skalierung notwendig: - Load Balancer vor dem Frontend & Backend - Mehrere Instanzen für Node.js-Scanner (Microservices) - Automatisches Skalieren mit Kubernetes oder AWS ECS - Separate Datenbank-Cluster (Read-Replica) - CDN für statische Inhalte - Erweitertes Monitoring (Prometheus, Grafana) - Logging-System (ELK-Stack)

Schicht	Technologie	Erweiterung für 1.000.000 Nutzer
Frontend	Next.js + CDN	Globale Verteilung & Caching
Backend	Node.js Microservices	Horizontale Skalierung & Queue-Verarbeitung
Orchestrierung	Kubernetes / Docker Swarm	Automatisches Skalieren
Datenbank	PostgreSQL Cluster	Read Replicas & Sharding
Queue & Cache	Redis/RabbitMQ Cluster	Mehrere Worker-Nodes

3. Schritt-für-Schritt Fahrplan

Phase 1 – MVP (0 bis 10.000 Nutzer) - Grundfunktion: Scans starten, Ergebnisse speichern, einfache UI - Next.js + Node.js Monolith (Backend integriert) - Single-Datenbank, einfacher Hosting (Vercel + kleiner Server) **Phase 2 – 100.000 Nutzer** - Backend in separaten Service auslagern - Job-Queue (BullMQ/RabbitMQ) integrieren - Redis Cache für Statusupdates hinzufügen - API Rate Limiting implementieren - Monitoring & Logging einbauen (z. B. Logtail, Sentry) **Phase 3 – 500.000**

Nutzer - Horizontale Skalierung: Mehrere Backend-Instanzen - Load Balancer einführen (AWS ELB, Nginx, Traefik) - Datenbank optimieren (Indexing, Read Replicas) - CDN für Frontend-Assets einrichten **Phase 4 – 1.000.000 Nutzer** - Kubernetes oder Docker-Swarm einführen für Auto-Scaling - Datenbank-Sharding oder Cluster (z. B. PostgreSQL Citus) - Microservices-Architektur für Scanner-Logik - Erweitertes Monitoring (Prometheus + Grafana) - Globales CDN und Multi-Region-Deployment