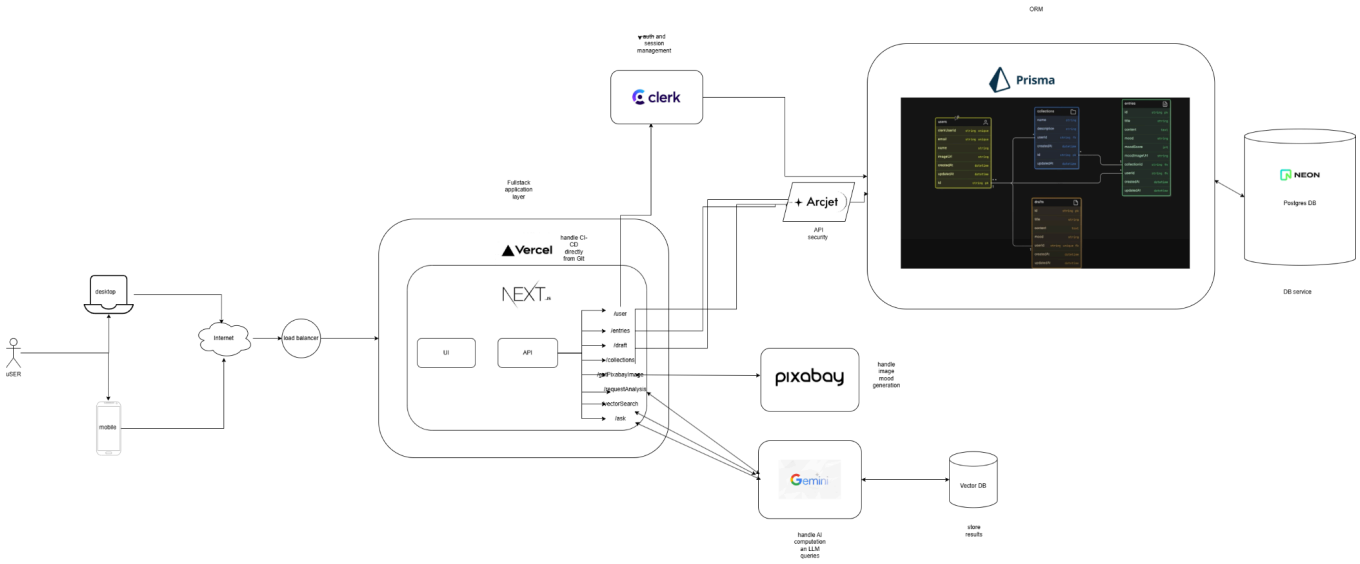
SHAJARA SYSTEM DESIGN DOCUMENT

Architecture diagram and explanation



Data model design and relationships

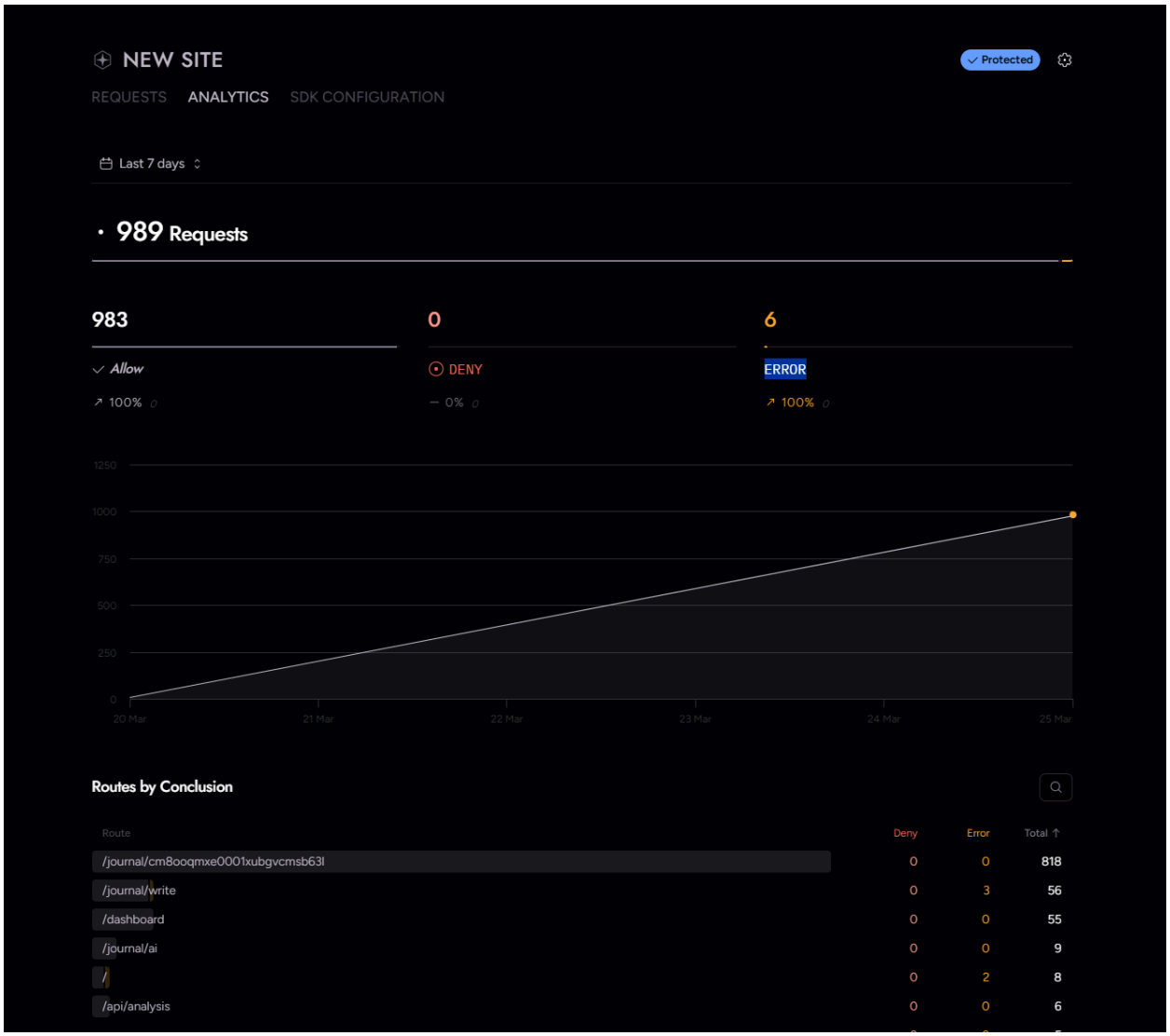
Security measures beyond basic authentication

I am using Arcjet library which I have integrated to offer all OWASP rules in turn offering protection against:

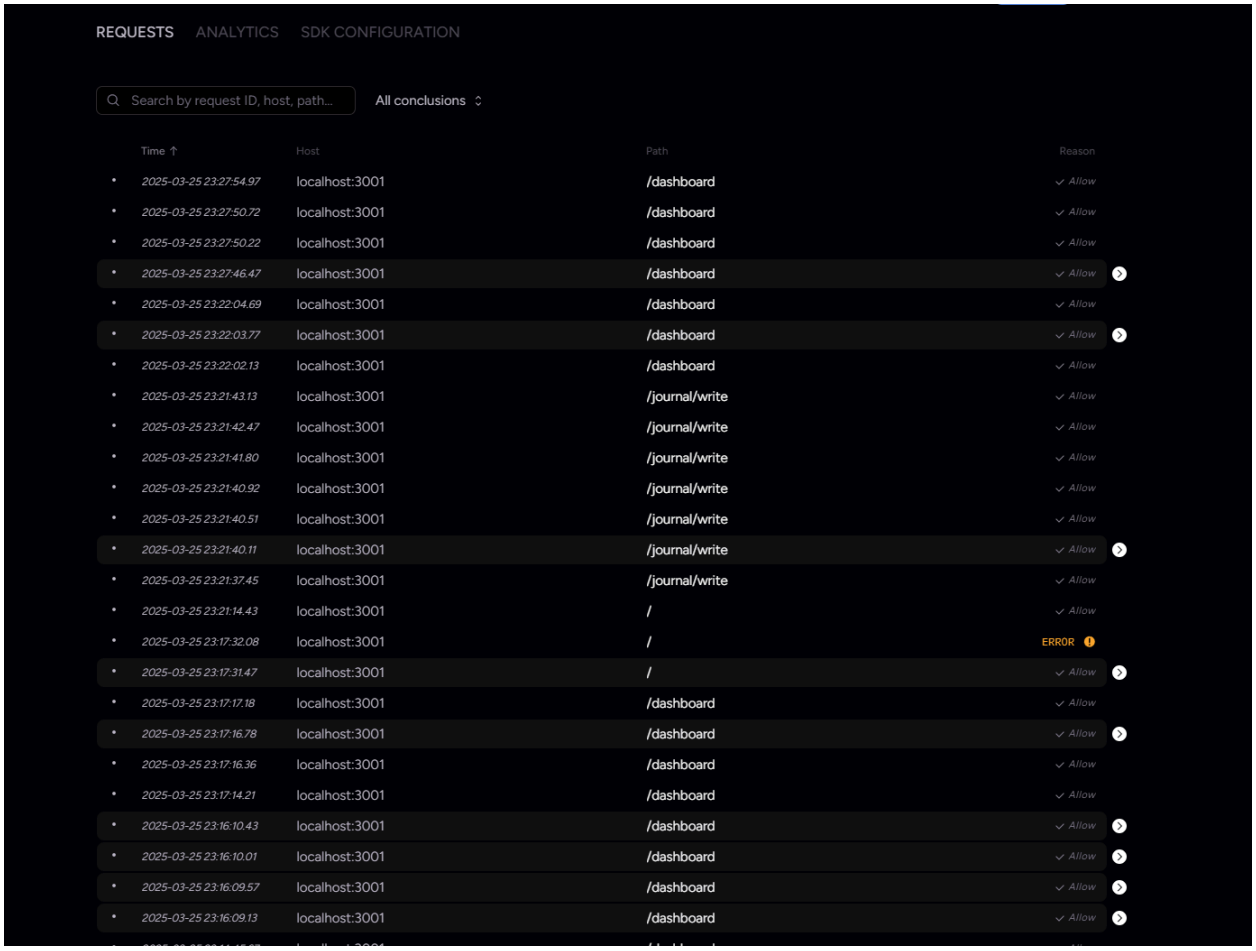
- SQL injection (SQLi)
- Cross-site scripting (XSS)
- Local file inclusion (LFI)
- Remote file inclusion (RFI)
- PHP code injection
- Java code injection
- HTTPoxy
- Shellshock
- Unix/Windows shell injection
- Session fixation

I have also implemented bot protection and rate limiting to APIs protection the application against DDOS attacks

Arcjet gives analytics



of all api calls which one was denied what was successful



Potential scaling challenges and solutions

- **How Ir design would scale to support 1M+ users**

Optimize My Codebase

- **Efficient Code**: Ensure my code is modular, reusable, and follows best practices.
- **Static Generation**: I would use Next.js's static site generation (SSG) for pages that don't require frequent updates.
- **Server-Side Rendering (SSR)**: I would optimize SSR for dynamic content to reduce server load.

Database Scaling

- **Read/Write Separation**: I would use a primary database for writes and replicas for reads.
- **Indexing**: I would optimize database queries with proper indexing.
- **Caching**: I would implement caching layers (e.g., Redis) to reduce database hits.

Backend Optimization

- **API Routes**: I would split heavy API routes into microservices if needed.
- **Load Balancing**: I would use load balancers to distribute traffic across multiple servers.
- **Asynchronous Processing**:I would offload heavy tasks to background workers or queues.

Frontend Performance

- **Code Splitting**:I would leverage Next.js's automatic code splitting to reduce initial load times.
- **CDN**: I would serve static assets via a Content Delivery Network (CDN) for faster delivery.
- **Lazy Loading**: I would implement lazy loading for images and components.

Infrastructure Scaling

- **Horizontal Scaling**:I would add more servers to handle increased traffic.
- **Auto-scaling**: I would use cloud providers like AWS, Azure, or GCP for auto-scaling based on demand.
- **Monitoring**: I would set up monitoring tools (e.g., Prometheus, Grafana) to track performance and bottlenecks.

Testing and Optimization

- **Load Testing**: I would simulate high traffic using tools like JMeter or Locust.
- **Performance Metric**s: I would continuously monitor metrics like response time, error rates, and server utilization.

Potential bottlenecks and how I would address them

1. Backend Server Bottlenecks

Issue: The backend server may struggle to handle concurrent requests as user traffic spikes.
Solution:

- **Load Balancing**: Deploy load balancers to distribute traffic across multiple servers.
- **Horizontal Scaling**: Spin up additional server instances to share the load.
- **Optimize API Routes**: Ensure API endpoints are efficient and minimize expensive operations.

2. Database Bottlenecks

Issue: Increased traffic can overwhelm Ir database, leading to latency and downtime.
Solution:

- **Read Replicas**: Use database replicas to split read queries from write queries.
- **Indexing**: Optimize database queries with proper indexing to speed up access.
- **Caching**: Implement caching solutions (e.g., Redis) for frequently accessed data.

3. Memory and CPU Bottlenecks

Issue: As traffic grows, memory and CPU resources can become strained.
Solution:

- **Auto-scaling**: Use cloud services to dynamically allocate resources based on traffic.
- **Profiling**: Continuously monitor CPU and memory usage and optimize processes consuming excess resources.

4. Network Bottlenecks

Issue: High traffic can saturate the network, leading to delays in data transfer.
Solution:

- **Content Delivery Network (CDN)**: Serve static assets via CDN to reduce network load.
- **Minify Files**: Minify JavaScript and CSS files to reduce bandwidth usage.
- **Compression**: Enable gzip or Brotli compression for faster data transfer.

5. Frontend Bottlenecks

Issue: A sluggish frontend can negatively affect user experience.
Solution:

- **Lazy Loading**: Load components and images on demand instead of upfront.
- **Code Splitting**: Use Next.js's code splitting features to minimize initial load time.
- **Client-side Caching**: Implement caching mechanisms on the client-side for static assets.

6. Deployment Bottlenecks

Issue: Deployment processes might become slow or error-prone as the app scales.
Solution:

- **Continuous Integration/Continuous Deployment (CI/CD)**: Automate deployment pipelines to ensure quick and reliable updates.
- **Blue-Green Deployments**: Minimize downtime during deployments by using blue-green strategies.

7. Monitoring Bottlenecks

Issue: Lack of visibility into app performance can hinder troubleshooting efforts.
Solution:

- **Monitoring Tools**: Implement tools like Prometheus, Grafana, or New Relic to track metrics such as latency, errors, and traffic.
- **Alerting Systems**: Set up alerts to notify I of anomalies or failures.

What components might need to be redesigned at scale

1. Application Architecture

- **Redesign from Monolith to Modular Architecture**: While a monolith application can work initially, I might need to split into microservices or modularize specific parts of the app (e.g., authentication, payments, user profiles) for scalability and independent development.
- **Stateless Servers**: Ensure that Ir servers are stateless, so scaling horizontally (adding more servers) becomes seamless.

2. Database

- **Sharding**: Divide the database into smaller, independent pieces based on user data (e.g., by geography, user ID ranges).
- **Distributed Databases**: Move to distributed databases like CockroachDB or Amazon DynamoDB to handle high traffic with high availability.
- **Read/Write Optimization**: Separate read operations (e.g., user feed queries) from write operations (e.g., posting new content) and use read replicas.

3. API and Backend

- **Rate Limiting and Throttling**: Protect my backend APIs from being overwhelmed by excessive or malicious requests.
- **GraphQL or gRPC**: Consider moving from REST to GraphQL or gRPC for more efficient data fetching and transport.

4. Caching and CDN

- **Caching Redesign**: Use caching solutions like Redis or Memcached for database queries, API responses, and rendered pages.
- **Edge Caching**: Leverage a Content Delivery Network (CDN) to cache static assets and pre-rendered pages closer to users.

5. Authentication and Authorization

- **Scalable Identity Providers**: Implement scalable identity solutions like AWS Cognito, Auth0, or custom OAuth solutions to handle millions of users.
- **Session Management**: Move sessions to a distributed store (e.g., Redis) rather than storing them on the server.

6. Frontend

- **Dynamic Content Optimization**: Redesign components to load content dynamically using APIs, reducing the need for server-side rendering (SSR) for every request.
- **Frontend State Management**: Use efficient state management libraries like Redux Toolkit or React Query to reduce unnecessary re-renders.

7. Deployment Infrastructure

- **CI/CD Pipeline**: Redesign the deployment pipeline to support faster, automated builds and rollbacks using blue-green or canary deployments.
- **Infrastructure as Code (IaC)**: Use tools like Terraform or AWS CloudFormation for provisioning infrastructure.

8. Monitoring and Observability

- **Enhanced Logging**: Move to distributed logging systems like Elasticsearch + Kibana for debugging issues across services.
- **Tracing**: Implement distributed tracing (e.g., with OpenTelemetry) to monitor request flow and identify bottlenecks.