Approach the problem statement THOUGHTFULLY , SCALABLE WELL-STRUCTURED, Clean code.

DB design and setup = mysql

Ui = good looking , interactive , cheking the inputs

Review = coding standards ; Modularity ; Logical Thinking ; Frontend design ; Database design , performance , scalability , usability ; Debugging skills

Approach to problem statement , coding patterns , Module structure , Attention to detail

**1️⃣ Database Level**

* **Pagination** → Never fetch huge datasets at once; always use LIMIT and OFFSET (or keyset pagination for better performance).
* **Indexes** → Add indexes on frequently queried columns (WHERE, JOIN, ORDER BY fields).
* **Connection Pooling** → Avoid creating a new DB connection per request; use a pool (e.g., MySQL pool in Node.js).
* **Read/Write Splitting** → Use a **primary DB** for writes and **replica DBs** for reads when traffic grows.
* **Caching** → Use **Redis** or Memcached for results that don’t change often.

**2️⃣ Server Level**

* **Stateless Servers** → Don’t store important data (like sessions) in memory; store in DB or Redis so any server can handle requests.
* **Load Balancer** → Distribute traffic across multiple servers (Nginx, HAProxy, AWS ELB, etc.).
* **Horizontal Scaling** → Add more servers instead of relying on a single powerful one.
* **Rate Limiting** → Prevent abuse by limiting requests per IP/user.
* **Compression** → Use Gzip/Brotli to reduce payload sizes.

**3️⃣ Frontend & API Design**

* **API Pagination / Filtering** → Never return 1M rows in a single request.
* **Lazy Loading / Infinite Scroll** → Load more data only when the user scrolls.
* **CDN for Assets** → Host static files (CSS, JS, images) on a CDN like Cloudflare, AWS CloudFront.
* **Minification & Bundling** → Reduce file sizes for faster loads.

**4️⃣ Networking Layer**

* **DNS Load Balancing** → Distribute users geographically.
* **SSL Termination** → Let the load balancer handle HTTPS to reduce server CPU load.
* **Geolocation Routing** → Send users to the nearest server.

**5️⃣ Monitoring & Recovery**

* **Logging** → Centralized logging (ELK stack, Grafana Loki).
* **Health Checks** → Load balancer removes unhealthy servers automatically.
* **Auto Scaling** → Cloud providers like AWS, GCP, Azure can add/remove servers based on traffic.
* **Backups & Disaster Recovery** → Frequent database backups.

**⚡ Example Scaling Stack**

1. **Frontend:** React/Vite served via CDN
2. **API Layer:** Multiple Node.js servers behind Nginx load balancer
3. **Database:** MySQL with read replicas + Redis cache
4. **Infrastructure:** Docker + Kubernetes for scaling
5. **Monitoring:** Prometheus + Grafana + alerts