Q6: System Scalability

The system employs several strategies to ensure smooth performance even with millions of simultaneous users:

1. **Horizontal Scaling Strategies:**
   a. **Microservices Architecture**
      - Service-specific scaling based on demand patterns (e.g., scaling Response Service during survey campaigns).
      - Independent technology stack optimization for each microservice based on its specific requirements.
      - Isolated failure domains prevent system-wide outages when a single service fails.
      - Independent deployment cycles allowing updates without complete system downtime.
      - Resource allocation optimization with smaller services requiring fewer resources than monolithic applications.

   b. **Stateless Service Design**
      - All core services (*Authentication*, *User*, *Document*, *Distribution*, and *Response* Services) are designed to be stateless, allowing for seamless horizontal scaling.
      - New service instances can be dynamically added as load increases without any service reconfiguration.
      - Each service can be scaled independently based on its specific resource demands.

   c. **Containerized Deployment**
      - All services are containerized, enabling easy deployment and scaling across the infrastructure.
      - Container orchestration platforms (like Kubernetes) can automatically scale services based on CPU, memory usage, or custom metrics.
      - Auto-scaling policies can be defined to proactively increase capacity before peak times.

   d. **Load Balancing**
      - Load balancers distribute traffic across multiple service instances.
      - Sticky sessions are implemented specifically for WebSocket connections to maintain consistency during real-time collaboration while allowing horizontal scaling.
      - Health checks ensure traffic is only routed to healthy instances.

2. **Database Scaling Strategies**
   a. **MongoDB Sharded Cluster**
      - Document data is stored in a MongoDB sharded cluster that horizontally scales by distributing data across multiple shards.
      - Sharding is based on document IDs, ensuring even distribution.
      - The sharded architecture allows for handling large volumes of CRDT operations and document storage.

   b. **PostgreSQL Scaling**
      - User data, permissions, and responses utilize PostgreSQL with read replicas to handle high read loads.
      - Connection pooling minimizes the overhead of establishing database connections.
      - Vertical scaling can be applied for write-heavy operations while read replicas handle query load.

   c. **Caching Layer with Redis**
      - Redis caching significantly reduces database load by storing frequently accessed document snapshots.
      - Distributed Redis clusters ensure cache performance scales horizontally.
      - Time-to-live (TTL) policies prevent cache bloat and memory exhaustion.

3. **Real-time Collaboration Scaling**
   a. **WebSocket Connection Management**
      - The *Real-time Collaboration* Service uses a hybrid approach to limit WebSocket message frequency:
        - Batching changes based on typing pauses (200ms).
        - Regular interval safety net (2 seconds).
        - Character threshold triggers (20 characters).
      - This prevents overwhelming the system during high-frequency editing.
      - The system uses sticky sessions for WebSocket connections which ensures that a client remains connected to the same instance of the *Real-time Collaboration* Service.

   b. **CRDT-Based Collaboration**
      - CRDT operations are inherently scalable as they don't require global locking.
      - The system uses distributed locks only for the minimal time needed to store operations.
      - Version vectors track causality without requiring a central coordination point.

   c. **Asynchronous Processing with Kafka**
      - Kafka decouples CRDT operation broadcast from persistence, allowing the system to handle operation spikes.
      - Kafka's partitioning enables parallel processing of operations.
      - The Kafka Consumer Service scales based on partition count to handle high volumes.

4. **Peak Load Management**
   a. **Rate Limiting and Throttling**
      - API Gateway implements rate limiting to prevent abuse and ensure fair resource allocation.
      - Throttling policies can be applied to non-critical operations during extreme peak loads.

   b. **Graceful Degradation**
      - Non-essential features degrade gracefully under extreme load (e.g., real-time analytics may update less frequently).
      - Critical operations (like saving form responses) are prioritized over less critical ones.

   c. **Regional Deployment**
      - Services can be deployed across multiple geographic regions to distribute load.
      - Users are directed to the closest region to minimize latency and balance global traffic.

5. **Monitoring and Predictive Scaling**
   a. **Comprehensive Monitoring**
      - Real-time monitoring of system performance metrics enables rapid identification of bottlenecks.
      - Alerts trigger when predefined thresholds are approached.

   b. **Predictive Auto-scaling**
      - Historical usage patterns inform predictive scaling policies.
      - The system can pre-scale before anticipated peak times (e.g., when large organizations schedule form distributions).

   c. **Load Testing**
      - Regular load testing simulates millions of concurrent users to identify scaling limitations before they affect real users.
      - Performance bottlenecks are addressed proactively rather than reactively.