Task Management System - Architecture Design Document

1. System Overview

The Task Management System is designed to handle millions of tasks and concurrent users while maintaining high availability and performance. The architecture follows a microservices-oriented approach with emphasis on scalability, security, and maintainability.

2. High-Level Architecture Components

2.1 Application Layer

- **Load Balancer Tier**
  - Uses AWS Application Load Balancer or NGINX for distributing traffic
  - Implements SSL termination
  - Health check monitoring for backend services
  - Automatic failover capabilities
- **API Gateway**
  - Rate limiting and throttling
  - Request validation and transformation
  - API versioning support
  - Traffic management and routing
- **Application Servers**
  - FastAPI instances running in containers
  - Horizontally scalable
  - Asynchronous request processing
  - Auto-scaling based on CPU/Memory metrics

2.2 Database Architecture

- **Primary Database (PostgreSQL)**
  - Master-slave replication setup
  - Read replicas for scaling read operations
  - Partitioning for tasks table based on creation date
  - Regular backup and point-in-time recovery

2.3 Authentication & Authorization

- **Keycloak Integration**
    - Centralized identity management
    - OAuth2/OpenID Connect support
    - Role-based access control (RBAC)
    - Multi-factor authentication support

```
# Authentication Middleware
async def authenticate_request(request: Request, keycloak:
FastAPIKeycloak = Depends(get_keycloak)):
    try:
        token = request.headers["Authorization"].split(" ")[1]
        user_info = keycloak.decode_token(token)
        return user_info
    except Exception:
        raise HTTPException(status_code=401, detail="Invalid
authentication credentials")
```

2.4 Caching Layer

- **Redis Cache**
    - Frequently accessed task data
    - Session management
    - Rate limiting information
    - Distributed locking mechanism

2.5 Monitoring & Observability

- **Logging System**
    - ELK Stack (Elasticsearch, Logstash, Kibana)
    - Structured logging format
    - Log aggregation and analysis
- **Metrics & Monitoring**
    - Prometheus for metrics collection
    - Grafana for visualization
    - Custom dashboards for business metrics

3. Scalability Strategy

3.1 Horizontal Scaling

- Docker containers orchestrated by Kubernetes
- Auto-scaling based on:
    - CPU utilization (target: 70%)
    - Memory usage (target: 80%)
    - Request queue length
    - Custom metrics

3.2 Database Scaling

- Read replicas for read-heavy operations
- Connection pooling using PgBouncer
- Query optimization and indexing
- Data partitioning strategy

```
# Database Connection Pool Configuration
DATABASE_CONFIG = {
    "pool_size": 20,
    "max_overflow": 10,
    "pool_timeout": 30,
    "pool_recycle": 1800,
}
```

3.3 Caching Strategy

- Implement cache-aside pattern
- Cache invalidation using event-driven approach
- TTL-based cache expiry
- Cache warming for critical data

3.4 Reliable Message processing

- Use Rabbitmq for durable message queues
- Persistent messages
- Retry mechanism with exponential backoff
- Dead letter queue for failed messages

4. Security Architecture

4.1 Authentication Flow

1. Client requests access token from Keycloak
2. Keycloak validates credentials and issues JWT
3. Client includes JWT in API requests
4. API validates token and authorizes requests

4.2 Security Measures

- SSL/TLS encryption for all communications
- Regular security audits and penetration testing
- Input validation and sanitization
- Rate limiting and DDoS protection

5. Performance Optimization

5.1 API Optimization

- Response compression
- Efficient pagination
- Asynchronous processing for long-running tasks
- Request batching where applicable

```
# Pagination Implementation
@app.get("/tasks/")
async def list_tasks(
    page: int = Query(1, gt=0),
    page_size: int = Query(10, le=100),
    db: AsyncSession = Depends(get_db)
):
    skip = (page - 1) * page_size
    query = select(Task).offset(skip).limit(page_size)
    result = await db.execute(query)
    return result.scalars().all()
```

5.2 Database Optimization

- Proper indexing strategy
- Query optimization
- Connection pooling
- Regular maintenance and vacuum

6. Deployment Strategy

6.1 Infrastructure as Code

- Docker Compose configuration for local development
- FastAPI application container
- PostgreSQL database container
- Redis cache container
- RabbitMQ message queue container
- NGINX reverse proxy container

6.2 Docker Swarm Setup

- Node configuration
    - Manager nodes for orchestration
    - Worker nodes for application hosting
- Service deployment strategies
    - Rolling updates
    - Automatic failover
    - Load balancing

6.3 CI/CD Pipeline

- Automated testing pipeline
- Unit tests
- Integration tests
- Code quality checks
- Image building and pushing
- Security scanning
- Automated deployments

This architecture design provides a robust foundation for a highly scalable Task Management System while maintaining security, performance, and reliability. The modular approach allows for easy updates and maintenance while supporting future growth and feature additions.