

Analysis of Fault-Tolerance and Resilience in the Distributed Dentist Appointment System DentiSmile+

Group 7

Yaroslav Ursul

Andre Ibrahim

Mahmoud Khalafallah

Ahmet Yavuz Kalkan

Maksym Matsuhryia

Date: 2025-01-12

Overview

The dentist appointment booking system for Gothenburg residents leverages a distributed architecture incorporating the Publish/Subscribe pattern via MQTT and Microservices with a Client frontend architectural style that are suited for distributed systems. This document identifies and analyzes the single points of failure (SPOFs) through stress-testing the DentiSmile+ microservices, discusses their implications, and proposes detailed mitigation strategies while drawing on principles from lectures and relevant literature. The reasoning behind usage of diagrams is to visualize possible system shortcomings and better clarify it for the customer and stakeholders.

Logs

Date	Identified bottleneck	Description	Mitigation Strategy
2024-11-25	MQTT broker	Our team quickly identified the need for multiple brokers in case one goes down. This mirrors our decision to explicitly include it early on in the system architecture diagram. The changes were made after Milestone 3 (2025/01)	Create multiple instances of the MQTT + use a load balancer to further increase system reliability
2024-12-20	Database	The database was also identified as a bottleneck, if it goes down, the system will too. We made sure to implement database redundancy for Appointment Service and use different databases for each service	Use multiple Databases for services, and database instances for appointment service to increase system reliability

Stress-Testing

Stress-testing the microservices of the system is important to ensure it meets the required performance standards under heavy load while maintaining fault-tolerance and resilience. This analysis evaluates the system’s behavior during high concurrency, identifies bottlenecks, and provides mitigation strategies for single points of failure (SPOFs). Apache JMeter was utilized to simulate concurrent users attempting to book dentist appointments.

Stress-Testing Objectives

1. Determine System Capacity — Estimate how many concurrent users the system can handle under normal and peak loads.
2. Identify Weak Points — Assess system components under heavy load.
3. Measure Availability — Validate the system's ability to maintain reasonable availability during stress.
4. Evaluate Scalability — Test system scaling mechanisms like load balancing and database optimizations.

Test Setup

The tools that we used are: Apache JMeter for stress-testing, Mosquitto for the MQTT broker, MQTT Explorer to verify publishing and subscriptions, and MongoDB for databases.

Test Plan:

1. Simulate concurrent booking attempts from 1000 users.
2. Each user requests slots using generated time intervals from a pre-generated .CSV file.
3. Publish appointment messages to the MQTT broker.
4. Verify message delivery, response times, and database consistency.

Test Data:

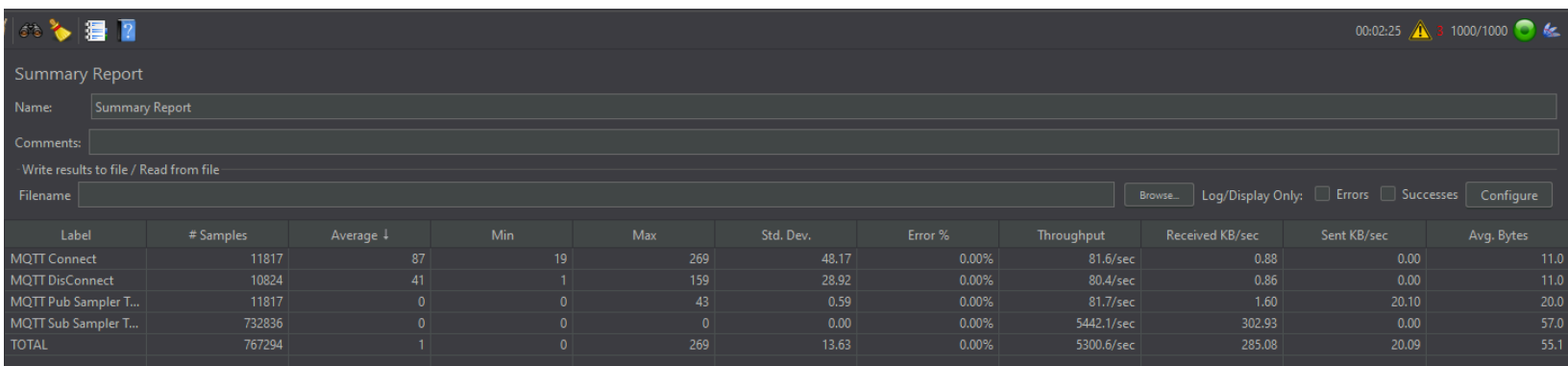
The `slots.csv` files were pre-generated using a custom python script. They contain: `dentistId`, `clinicId`, `clinicName`, `patientId`, `startTime`, `endTime`, `status`

Slots were loaded dynamically into the system to ensure unique bookings.

Results

1. Performance Under Normal Load: The system was able to handle 1000 users concurrently, with an average response time of 10ms for publishing/subscriptions and 87/41ms for connecting and disconnecting the MQTT broker, respectively. Subscription request throughput of 5442 requests/second.
2. Performance Under Heavy Load: With 10000 users attempting concurrent bookings, system performance degraded:
 - The MQTT broker disconnected.
 - MQTT message delivery failure rate increased to 80% due to broker overload.
3. Identified Bottlenecks:
 - MQTT Broker: The single broker instance became a SPOF under high load.
 - Database: The appointment service database faced significant strain during slot queries and write operations.

- Scalability: Limited horizontal scaling of microservices and lack of caching exacerbated delays.



Label	# Samples	Average ↓	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
MQTT Connect	11817	87	19	269	48.17	0.00%	81.6/sec	0.88	0.00	11.0
MQTT Disconnect	10824	41	1	159	28.92	0.00%	80.4/sec	0.86	0.00	11.0
MQTT Pub Sampler T...	11817	0	0	43	0.59	0.00%	81.7/sec	1.60	20.10	20.0
MQTT Sub Sampler T...	732836	0	0	0	0.00	0.00%	5442.1/sec	302.93	0.00	57.0
TOTAL	767294	1	0	269	13.63	0.00%	5300.6/sec	285.08	20.09	55.1

Figure 1. Stress-testing Summary Report

Identified SPOFs and Challenges

MQTT broker — The broker facilitates communication between entities. Its failure would result in a complete breakdown of system messaging, halting operations.

Databases — As the repository of all critical data, the database's unavailability disrupts access to appointment slots and bookings.

Fault-Tolerance Concerns and mitigation strategies

There is currently no implemented redundancy for our MQTT broker or databases, making them susceptible to outages. A failure in either of the components would propagate through the system, amplifying the impact and potentially rendering the system useless. Possible mitigation strategies include deploying a redundant clustered MQTT broker architecture, such as Mosquitto clusters, or implementing load balancers and circuit breakers, which can distribute the workload and lower the risk of a single broker failure. Lecture notes emphasize that decoupled systems architectures that use Publish/Subscribe benefit from this [\[1\]](#). Implementing replica sets for the databases, as supported by MongoDB, would provide higher availability and fault-tolerance by maintaining multiple copies of the data, but would ultimately result in additional complexity.

Identified Weaknesses

Operations such as slot retrieval are read-heavy and depend on real-time database access. A failure in the database affects the entire booking process. Furthermore, there is no provision for caching or load distribution, which increases stress on the primary database during peak usage.

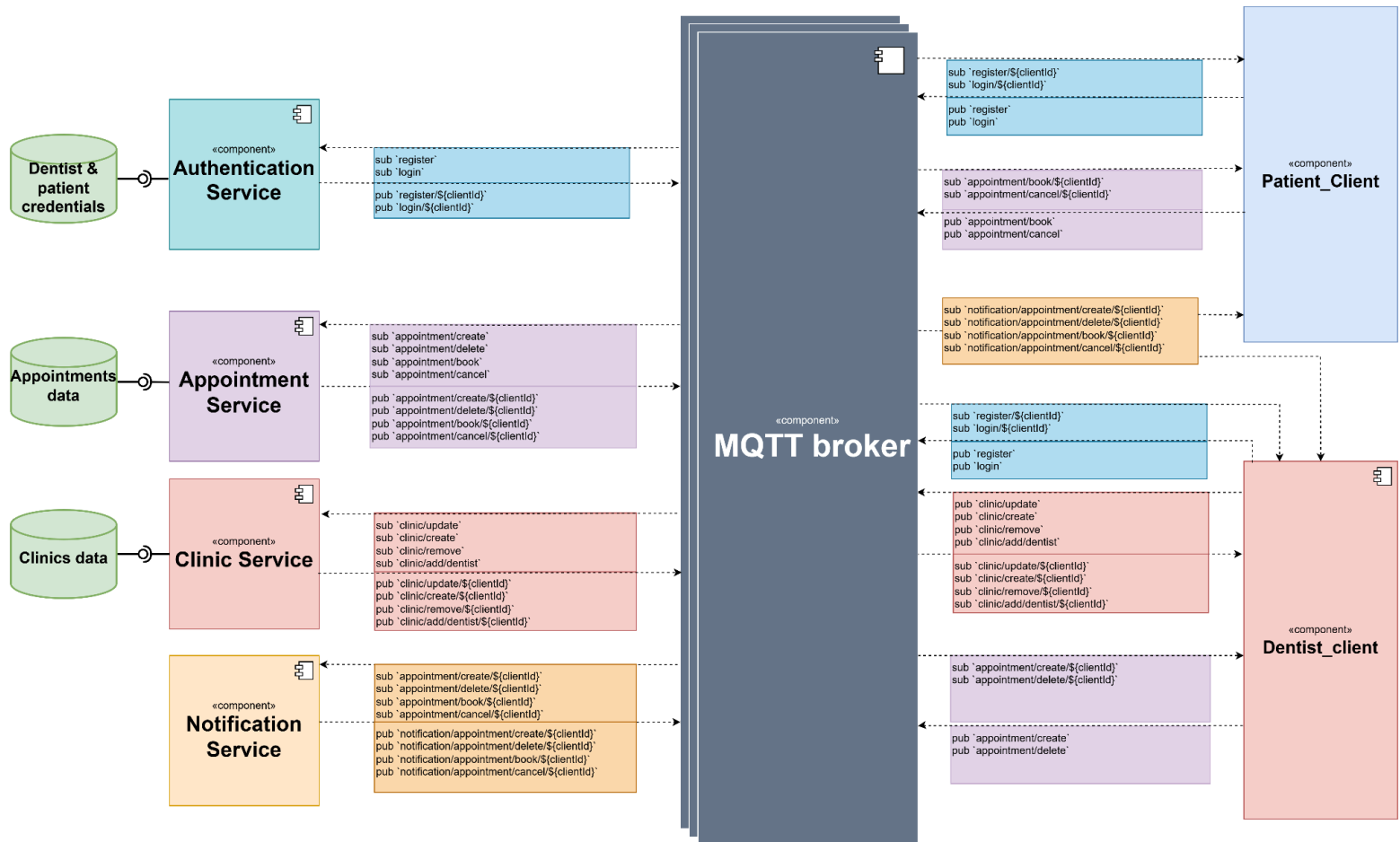
Proposed Improvements & Mitigation Strategies

Database Read Replicas — Using read replicas can offload traffic from the primary database, thereby reducing the risk of overload, even if read replicas are not full system back-ups.

Caching Layer — Incorporating a distributed caching solution (e.g., Redis) for frequently accessed data, such as available slots, would improve response times and alleviate database strain.

Architectural Diagram Analysis

The architectural diagram presents the deployment of clients (Web GUI and Dentist Tool), middleware using MQTT, and the database for storage.



Observed Vulnerabilities in System Architecture Diagram

- The middleware relies on a single MQTT broker instance, presenting a critical SPOF.
- The database has no failover mechanisms, leaving the system vulnerable to downtime.

Mitigation Strategies

Distributed Middleware — Deploying multiple MQTT brokers (as depicted in the diagram by multiple instances) behind a load balancer can provide fault-tolerance and improve scalability. Tools such as Kubernetes can automate the recovery and scaling of brokers.

Database Solutions — Using high-availability databases (possibly locally-hosted databases as suggested by teacher Hans-Martin Heyn), such as PostgreSQL with automated failover, ensures continuous access to critical data. Lectures on resilience [2] emphasize the importance of redundancy in mitigating SPOFs.

Deployment Diagram Analysis

This diagram (Figure 4) depicts workflows for booking slots and managing cancellations, with interactions mediated by the MQTT broker.

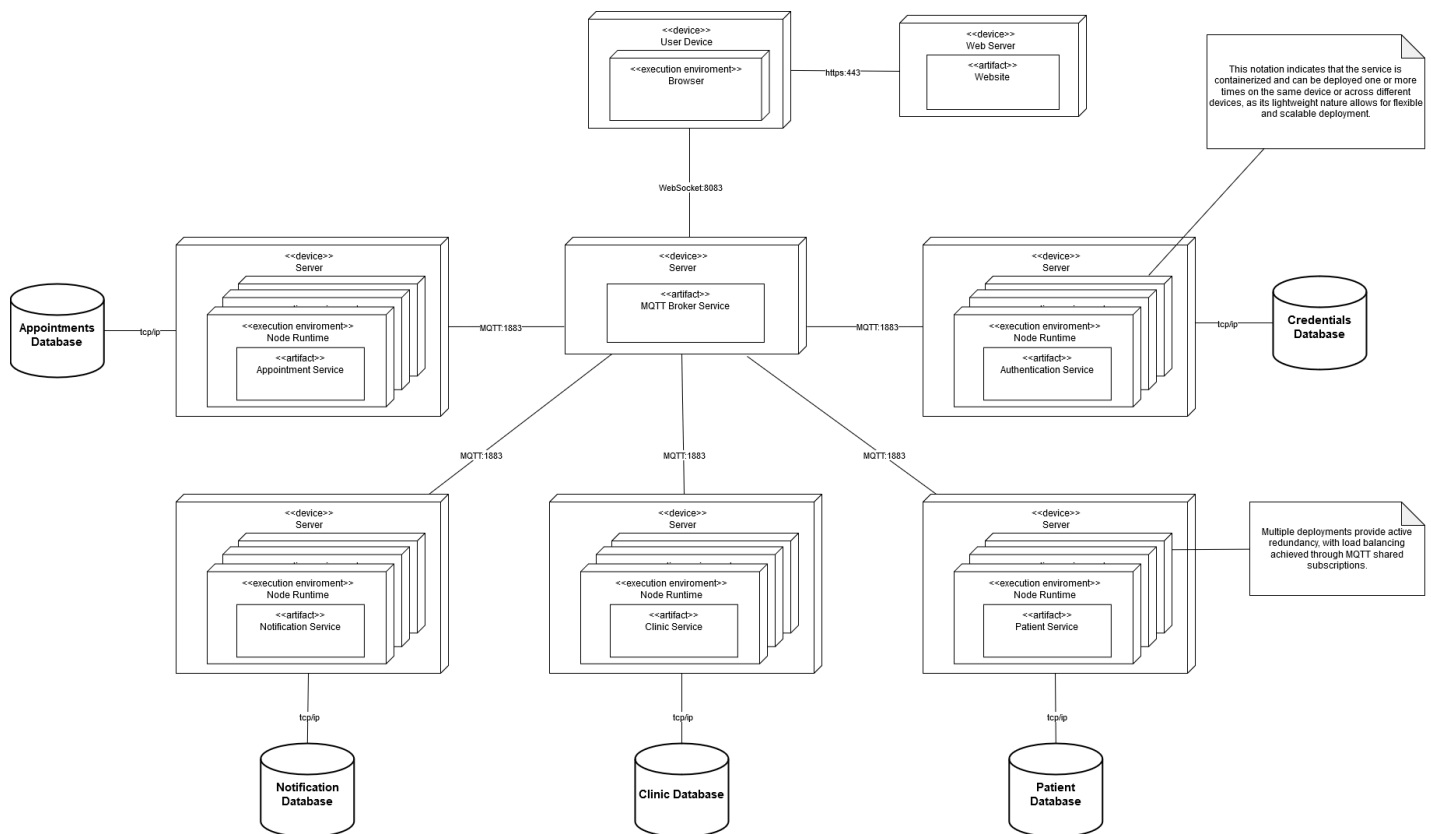


Figure 4. Deployment Diagram

Observed Vulnerabilities in Deployment Diagram

- Once again, the obvious dependency on a single broker increases the likelihood of failures and disrupting operations.

- Booking and cancellation processes lack safeguards to recover from transactional inconsistencies caused by outages (lack of back-ups).

Enhancements

Reliable Messaging — Utilizing MQTT's QoS 2 level guarantees exactly-once delivery of critical messages like booking confirmations. This approach mitigates risks associated with message duplication or loss [\[1\]](#).

Transactional Logging — Adding transactional logs ensures that partially completed operations can be retried or rolled back, maintaining system consistency.

Sequence Diagram Breakdown

The sequence diagram (Figure 5) illustrates the use case of a new user booking an appointment. Key interactions include user authentication, availability checking, and booking confirmation.

Account Creation and Authentication

SPOFs

Authentication Service — If down, the user cannot log in or create an account.

Authentication Database Server — A failure here halts account verification.

Mitigation

- Implement a secondary authentication service instance and failover mechanisms.
- Use database replication (e.g., PostgreSQL streaming replication).

Request Available Appointments

SPOFs

Appointment Service — If unavailable, no data can be retrieved.

Appointment Database Server — A failure disrupts the retrieval of slot data.

Mitigation

- Use distributed caching (e.g., Redis) to store frequently accessed appointment data.
- Horizontal scaling for the Appointment Service.

Use case: New user
booking an appointment

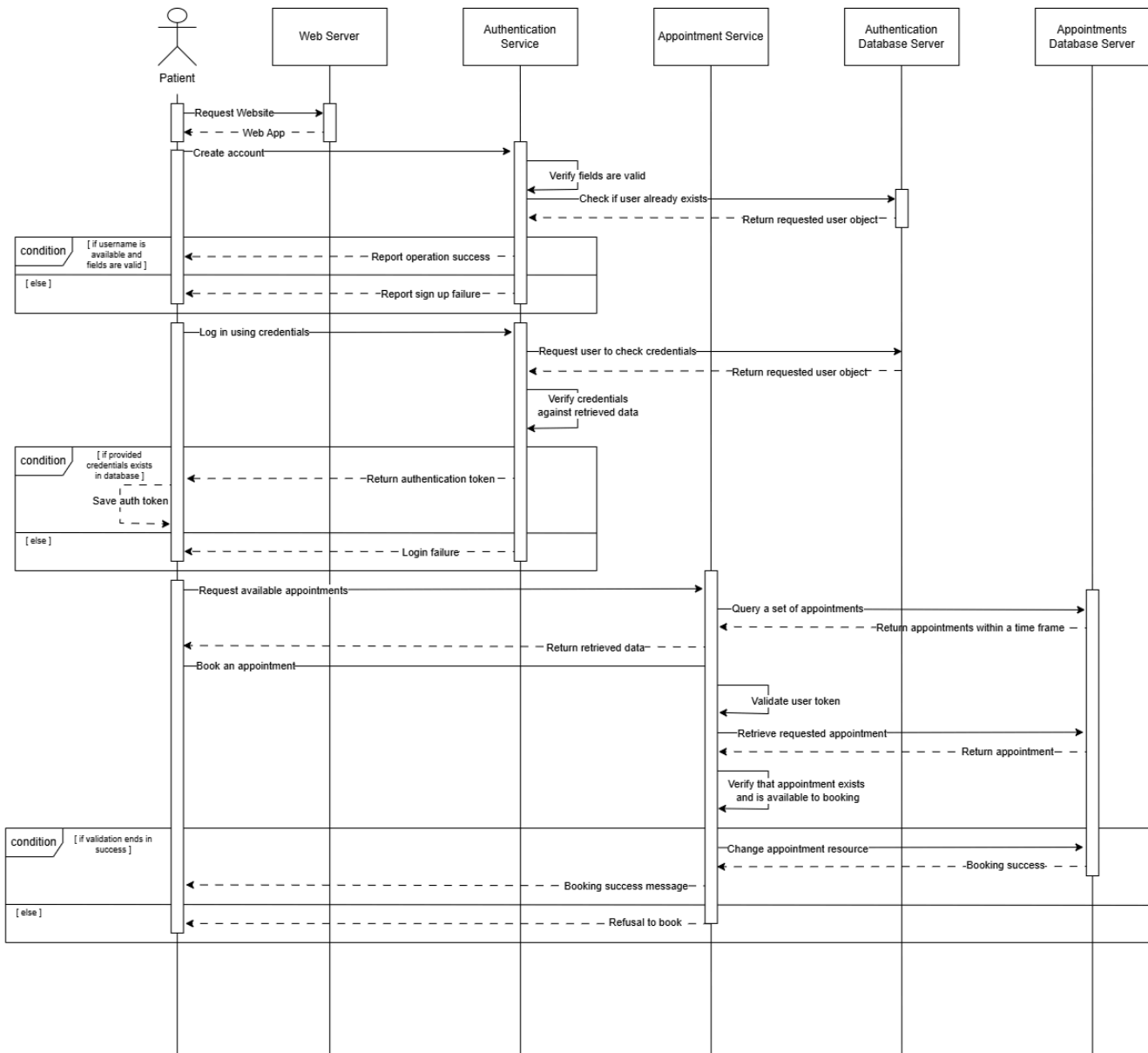


Figure 5. Sequence Diagram for a User Case Scenario (Booking an Appointment)

Booking an Appointment

SPOFs

Message Delivery via MQTT Broker — Any loss here impacts slot updates and booking confirmation.

Database Write Operations — Failure during booking confirmation can lead to inconsistencies.

Mitigation

- Set MQTT QoS to 2 (exactly-once delivery) for booking-critical messages [\[1\]\[2\]\[3\]](#)
- Implement transactional logging and distributed transactions (e.g., using Saga patterns).

Fault-Tolerance Analysis for Overall System

Main SPOFs and Challenges:

MQTT Broker

Central to publish/subscribe messaging. Failure halts all communications.

Proposed Solution — Use a clustered MQTT broker architecture with load balancing (e.g., Eclipse Mosquitto).

Databases

Critical for storing user, slot, and appointment data. Single-node failures disrupt the entire workflow.

Proposed Solution — Database replication (e.g., MongoDB replica sets or PostgreSQL replication) and geo-distributed clusters to ensure regional availability.

Additional Enhancements

Caching Layer — Reduce load on backend databases using caching systems like Redis for frequently queried data (e.g., available slots).

Load Balancing — Distribute user requests across redundant service nodes using tools like Nginx or AWS Elastic Load Balancing.

Monitoring and Alerts — Employ Prometheus and Grafana for real-time system health monitoring. [\[1\]\[4\]](#)

Scalability and Stress Management

To address the anticipated expansion of more users and clinics in Sweden:

Scaling Strategies

1. Horizontal Scaling

Deploy multiple service instances for both the Appointment and Authentication services to handle concurrent users.

2. Auto-Scaling and Predictive Load Management

Set up auto-scaling for critical components like MQTT brokers and backend services.

Integrate machine learning models to predict peak loads and pre-scale resources.

3. Database Partitioning/Sharding

Shard databases based on user locations or dentist clinics to distribute load effectively.

4. Stress Testing Tools

Build custom stress-testing scripts simulating thousands of concurrent users booking appointments. [\[2\]](#)[\[3\]](#)

5. Observability Tools

Real-time dashboards for admin views to track live metrics (e.g., booked slots, failed requests).

Future Resilience Enhancements

Long-Term Strategies

1. Geo-Redundancy

Deploy geographically distributed nodes for database and middleware.

2. Eventual Consistency Models

Transition to eventual consistency for slot availability in case of partial outages.

3. Disaster Recovery Plans

Backup MQTT brokers and databases in disaster recovery zones.

4. Decentralized Broker Network

Implement a decentralized MQTT broker network with federated communication.

Conclusion

This analysis identifies critical points of failure and presents immediate and long-term solutions for building a fault-tolerant and resilient distributed dental system for clinics in Sweden. Stress-testing and analyzing diagrams like the sequence, architectural, and deployment diagrams are integral views in uncovering bottlenecks and strategizing mitigation tactics. Iterative testing and architecture refinement will further bolster system robustness overtime.

References

1. Hans-Martin Heyn, DIT356 Mini Project: Distributed Systems Development, Lecture 3: *“Architectures of Distributed Systems”*, Chalmers and University of Gothenburg;
2. Hans-Martin Heyn, DIT356 Mini Project: Distributed Systems Development, Lecture 5: *“Scalability and Fault Tolerance”*, Chalmers and University of Gothenburg;
3. O’Reilly Media, *“Designing Data-Intensive Applications”* by Martin Kleppmann;
4. MQTT.org, *“MQTT Essentials: Features and Best Practices”*.