Scalability – Proof of Concept (PoC)

1. **Dizajn šeme baze podataka (konceptualni, logički ili fizički)**
   Dizajn šeme baze podataka je dat kroz konceptualni dijagram pomoću koga su generisani logički i fizički dijagram. U pratećem folderu nalaze se PowerDesigner projekti u kojima su kreirani konceptualni, logički i fizički dijagrami.

2. **Predlog strategije za particionisanje podataka**
   Upotrebom particionisanja povećava se skalabilnost i performanse aplikacije.
   Skladištenje podataka na jednom serveru veoma brzo može dovesti do hardverskog limita. Jedan od najčešće korišćenih pristupa jesu horizontalno i vertikalno particionisanje podataka. Kada je u pitanju horizontalno particionisanje, moguće je podeliti tabele sa velikim brojem entiteta možemo podeliti na više manjih po nekom ključu. U slučaju naše aplikacije to bi bila tabela sa rezervacijama za preuzimanje medicinske opreme, pa bismo mogli da podelimo tabelu na osnovu administratora kompanije. Id administratora kompanije se može dodeliti hash funkciji koja će odrediti koji server baze podataka će čuvati rezervacije koje pripadaju određenom administratoru kompanije. Takođe, moguće je izvršiti i particionisanje rezervacija i po vremenu njihovog nastanka. Rezervacije koje su se završile u prošlosti nemaju preveliki značaj. Kada je u pitanju vertikalno particionisanje, ponovo se možemo poslužiti primerom termina rezervacije. U jednoj particiji bi čuvali podatke kojima se ređe pristupa kao što su id korisnika koji je izvršio rezervaciju, da li je termin predefinisan i slično, dok bi u drugoj particiji čuvali početak i trajanje rezervacije. Drugoj particiji bi se češće pristupalo zbog provere da li se rezervacije preklapaju.
   Particionisanje podataka može u velikoj meri da poveća sigurnost naše aplikacije na taj način što bismo osetljive podatke mogli čuvati na posebnim serverima. Tako možemo mail-ove, korisnička imena i lozinke čuvati na posebnim serverima. Na taj način bi se i veličina upita smanjila, jer se ovi podaci najčešće koriste pri autentifikaciji korisnika.

3. Predlog strategije za replikaciju baze i obezbeđivanje otpornosti na greške

Za implementaciju je korištena PostgreSQL baza podataka, koja kao sastavni deo ima replikaciju. PostgreSQL se deli na *master* i *slave* bazu. Podaci bi se nalazili na master bazi i na nju bi se vršilo pisanje, dok bi se kopije nalazile na slave bazama. U sistemu bi se radila asinhrona sinhronizacija podataka između master i slave baza kako se mreža ne bi zagušivala. Ukoliko bi došlo do otkaza master baze, rad aplikacije se nastavlja korišćenjem slave baze. Na ovaj način je očuvana **otpornost na greške.** Takođe, ukoliko se dogodi neka prirodna nepogoda, podaci će biti sačuvani, jer imamo podatke na našim slave bazama.

4. Predlog strategije za keširanje podataka

Osnovni cilj keširanja podataka je poboljšanje brzine pristupa i korišćenja podataka, stoga predlog se zasniva prvenstveno na određivanju najčešće korištenih podataka. Vodeći se pretpostavkom da će se rezervacijama koje su se obavile davno dosta ređe pristupati, koristili bismo se strategijama *least recently used* i *least frequently used* čime bismo keširali podatke novijih datuma i koji se dosta češće koriste. Pošto je keš ograničenog kapaciteta, dolaskom novih podataka najstariji podaci će biti izbačeni.

5. Okvirna procena za hardverske resurse potrebne za skladištenje svih podataka u narednih 5 godina

Pretpostavke sistema:

- ukupan broj korisnika: 100 miliona
- broj rezervacija svih entiteta na mesečnom nivou je milion
- sistem mora biti skalabilan i visoko dostupan

Zauzetost memorije za skladištenje pojedinačnih entiteta u bazi:

- Address: 1 KB
- User: 2.5 KB
- Company: 0.5 KB
- Predefined Term: 0.3 KB
- Term: 0.58 KB

Za 100 miliona korisnika potrebna memorija iznosi 320 GB.
Za milion rezervacija mesečno, tokom godinu dana potrebno je 3.6 GB za predefinisane termine i 7 GB za klasične termine.
Ukupno, za održavanje sistema u trajanju od 5 godina potrebno je približno 400 GB.

6. Predlog strategije za postavljanje load balansera

Ideja upotrebe load balancer-a jeste da opterećenje prenesemo na više servera i da jedan server ne obrađuje sve zahteve koje mu šalju klijenti. Stoga se postavlja load balancer između klijenta i aplikativnog servera. Prema nekim istraživanjima algoritam za load balancing koji daje dobre rezultate jeste *Least Pending Requests.* Ideja algoritma je da se prati broj zahteva koje određeni server ima pred sobom da obavi. Ukoliko je broj zahteva veći, to će i vreme za njihovo izvršenje biti veće. U realnom vremenu se prati broj zahteva koji svaki server treba da obradi i kandidat za novopristigli zahtev je onaj server koji ima najkraći red zahteva. Algoritam može efikasno da se nosi sa vremenskim ograničenjima za zahteve. Automatski prepoznaje sve činioce koji će dovesti do produžetka redova zahteva, te nove zahteve prosleđuje na manje opterećene servere.

7. Predlog koje operacije korisnika treba nadgledati u cilju poboljšanja sistema

Poboljšanje aplikacije bi se moglo ostvariti pomoću paging-a, jer bismo na ovaj način ubrzali dobavljanje podataka tako što svi podaci ne bi bili dobavljeni istovremeno, već bismo dobavljali samo određenu količinu podataka. Takođe, praćenje logovanja bi bilo značajno u cilju podsticanja korisnika na datu aktivnost. Na primer, ukoliko je prošlo određeno vreme, a korisnik se nije prijavio na sistem, aplikacija bi mogla da pošalje mejl u kome ga obaveštava o najnovijim akcijama koje je propustio. Slično, ukoliko se određena korisnička rezervacija završila, a korisnik je nije ocenio, isti bi mogao da dobije notifikaciju koja bi ga periodično podsećala da to može uraditi.

8. **Kompletan crtež dizajna predložene arhitekture (aplikativni serveri, serveri baza, serveri za keširanje, itd)**