1. Functional Requirements

The system must support the following core functions:

Customer-facing requirements

- Customers must be able to browse available services and view provider profiles.
- Customers must be able to request a booking for a specific service, time, and location.
- Customers must be able to receive real-time updates when a provider accepts or rejects a request.
- Customers must be able to complete payments through the integrated payment system.
- Customers must be notified if the provider cancels or does not respond in time.

Provider-facing requirements

- Providers must be able to create and manage profiles.
- Providers must be able to list services with descriptions, pricing, availability, and images.
- Providers must be able to receive real-time booking requests.
- Providers must be able to accept or reject requests.

System requirements

- The system must create and store bookings reliably.
- The system must assign and notify providers in real-time.
- The system must handle payment creation and refunds.
- The system must maintain complete payment and booking history.
- The system must enforce authentication on all sensitive actions.

---

2. Non-Functional Requirements

Performance

- Most API responses should return within 200–500 ms.
- Real-time updates must be delivered with minimal latency (<1 second ideally).

Scalability

- The backend must handle increased numbers of concurrent provider requests and customer bookings.
- Search and booking flows should scale horizontally as traffic grows.

Availability

- The system should maintain high uptime, especially during peak hours.
- Critical functions (booking, provider acceptance, payments) must remain stable.

Security

- All network communication must use HTTPS.
- JWT validation must occur on every API request.
- Providers should only access their own data.
- API secrets must only exist on the backend.

Reliability

- Booking state transitions must be consistent and durable.
- Payment and refund flows must not produce inconsistent states.

Observability

- Logging should track all booking events, errors, and provider actions.
- Monitoring for WebSocket uptime, Stripe failures, and slow endpoints.
- Metrics should help diagnose provider response times and booking drop-offs.

---

3. Architectural Style Decision

Servio’s MVP uses a **modular monolith** architecture with a single Flask-based API Gateway and a set of internal modules (BookingService, SearchService, PaymentService).

Why a modular monolith?

- Faster to build and easier to maintain at MVP scale
- Reduced operational overhead compared to microservices
- All business logic remains in one codebase but separated by clear module boundaries
- Simple deployments match early-stage requirements
- Easier testing and debugging

Why not microservices yet?

- Microservices add significant complexity (orchestration, CI/CD, monitoring, distributed tracing)
- Too costly early on when traffic is small
- Splitting into services prematurely increases risk and slows development

Why Flask Gateway?

- Lightweight, simple, and perfect for request routing
- Easy integration with Stripe, PostgreSQL, Firebase, and WebSockets

When microservices make sense (future phases):

- If provider matching grows complex
- If real-time load increases significantly
- If provider analytics or high-scale search requires independent scaling

---

4. API Paradigm Decision

Servio uses **GraphQL** as the main API paradigm for the client-facing interface.

Why GraphQL?

Servio is a highly relational marketplace application with connected data between different enities

- A service and its provider

- Provider and their service with its availability dates

- Bookings with user, provider, service, slot, and payment

- Lists of services with basic info, categories, and provider ratings

Single-call nested data fetching

Instead of making 3–6 REST calls, the app can fetch everything in **one query**.

No over-fetching or under-fetching

Mobile apps benefit greatly from receiving *only* the fields they need.

Strong type system

Servio’s complex entities (users, providers, bookings, payments) map naturally to GraphQL types.

Excellent fit for scalable modular monolith

GraphQL resolvers align well with modular back-end structure:

- BookingService → booking resolvers
- SearchService → search resolvers
- PaymentService → payment resolvers

---

5. Several GraphQL query examples

```
query1 = '''
   query GetServices {
     services {
       id
       title
       price
       provider {
         id
         name
       }
     }
   }
'''

query2 = '''
   query ServiceAvailability($providerId: ID!) {
     services_by_pk(id: $providerId) {
       id
       name
       unavailable_dates {
         id
         start
         end
       }
     }
   }

'''

query3 = '''
   query GetBookings {
     bookings {
       id
       status
       service {
         title
         price
       }
       provider {
         name
       }
       slot {
         start_time
       }
       payment {
         amount
         status
       }
     }
   }
'''
```

⭐ 6. Authentication & Authorization

Servio uses **Firebase Authentication with JWT tokens**.

Authentication Flow

- Users (customers and providers) authenticate in App using Firebase Auth
- Firebase issues a signed JWT
- The client sends the JWT in `Authorization: Bearer` headers
- The API Gateway verifies the token on every request

Authorization

The system supports two main roles:

Customer

- Can browse services
- Can create bookings
- Can pay
- Can cancel a booking they created

Provider

- Can manage profile
- Can create service listings
- Can accept or reject bookings assigned to them

Unauthorized access attempts must be rejected with HTTP 401.

⭐ 7. Data Access & Patterns

Read Patterns

- Customers frequently read lists of services and provider profiles
- Providers read their upcoming bookings
- SearchService fetches availability and service categories
- High read-to-write ratio → PostgreSQL handles this well

Write Patterns

- Bookings and payments are write-heavy during peak times
- Provider status updates are frequent
- PaymentService writes transaction logs after Stripe responses

Caching Decisions

MVP:

- No Redis caching required
- Image delivery handled by Firebase Storage CDN

Phase I or later:

- Can introduce Redis for caching service lists or provider availability

Data Synchronization

- BookingService and PaymentService must update states atomically

- Stripe webhooks guarantee payment state completeness
- Provider availability updates must remain consistent with bookings