**Microservice Independence**

Each microservice (Order, Payment, Email, Production) runs independently and can be deployed or scaled separately without affecting other services.

**Communication Protocol**

All services communicate via REST APIs using HttpClientFactory. No message queue or asynchronous event bus is implemented in this version.

**Service Availability**

PaymentService, EmailService, and ProductionService are assumed to be available when OrderService calls them.

No retry or circuit breaker logic has been implemented at this stage.

**Configuration Management**

Service URLs and endpoints are configured in appsettings.json under ServiceEndpoints and injected into OrderService via IConfiguration.

**Standardized Response Format**

All APIs return data wrapped in ApiResponse<T> with Result, Status, and Message to ensure consistency.

**Database for OrderService**

OrderService uses an In-Memory Database for demo and testing purposes. Data will be lost when the service restarts. In production, a persistent database (e.g., SQL Server) should be used.

**Sample Data for Testing**

SampleDataSeeder initializes the database with predefined test cases:

Case 1: OrderStatus = Pending

Case 2: TotalAmount % 2 != 0

Case 3: OrderStatus != Pending

Case 3: Total quantity > 100

**External Service Simulation**

Payment, Email, and Production services are mocked with minimal controllers that return success or failure responses based on request logic.

**Authentication & Authorization**

No authentication or authorization mechanism is implemented at this stage. All endpoints are assumed to be publicly accessible within a trusted environment.

**Email Sending Logic**

EmailService does not send real emails. It only simulates sending by returning a success response.

**Order Validation**

Order checkout is allowed only if Status == Pending.

Orders are linked to a CustomerId and verified before processing.

**Error Handling**

If any external service (Payment, Production, Email) fails, the process stops, and the order status is updated accordingly:

PaymentFailed if payment fails

ProductionFailed if production service fails

Email failures do not block order completion but should be logged.

**Performance & Scalability**

Performance optimization and scaling strategies (load balancing, service discovery, caching) are out of scope for this initial implementation.

**Network Latency and Timeouts**

Network latency between services is assumed to be minimal. No timeout or fallback logic is implemented.

**Environment**

All services are designed to run locally for development. Docker or container orchestration is optional for production deployment.