

Bounded Context, Context Map, dan Konteks Inti
II3160 - Teknologi Sistem Terintegrasi



Disusun Oleh:

Kenzie Raffa Ardhana - 18223127

Program Studi Sistem dan Teknologi Informasi
Sekolah Teknik Elektro dan Informatika - Institut Teknologi Bandung
Jl. Ganesha 10, Bandung 40132
2025

Daftar Isi

Daftar Isi.....	2
Daftar Gambar.....	5
Daftar Tabel.....	6
1. PENDAHULUAN.....	7
1.1 Tujuan Dokumen.....	7
1.2 Dasar Analisis.....	7
2. IDENTIFIKASI OF BOUNDED CONTEXTS.....	7
2.1 Reservation Context [CORE].....	7
2.2 Pricing Context [CORE].....	8
2.3 Inventory Context [SUPPORTING].....	10
2.4 Guest Context [SUPPORTING].....	11
2.5 Operations Context [SUPPORTING].....	12
2.6 Payment Context [STRATEGIC].....	13
3. CONTEXT MAP.....	14
3.1 Diagram Context Map.....	14
3.2 Relationship Patterns Detail.....	16
3.2.1 Reservation dengan Inventory (OHS/PL).....	16
3.2.2 Reservation dengan Pricing (ACL).....	17
3.2.3 Reservation dengan Guest (Customer/Supplier).....	18
3.2.4 Pricing dengan Operations (Conformist).....	20
3.2.5 Guest dengan Payment (ACL).....	22
3.2.6 Inventory dengan Operations (Shared Kernel).....	24
3.3 Context Map Summary Table.....	27
4. PEMILIHAN KONTEKS INTI.....	28
4.1 Konteks Inti yang Dipilih: RESERVATION CONTEXT [CORE].....	28
4.2 Kriteria Evaluasi.....	28
4.3 Analisis Komparatif.....	29
4.3.1 Analisis Komparatif.....	29
4.4 Justifikasi Final.....	30
5. STRATEGIC DESIGN - RESERVATION CONTEXT.....	31
5.1 Core Domain Model Overview.....	31
5.1.1 Aggregates.....	31
5.1.2 Value Objects.....	37
5.1.3 Domain Services.....	43
6. TACTICAL DESIGN - RESERVATION CONTEXT.....	47
6.1 Aggregate Identification.....	47
6.2 Identified Aggregates.....	48
6.3 Domain Events Strategy.....	50
6.4 Repository Implementation Strategy.....	51
7. STRATEGI IMPLEMENTASI.....	51
7.1 Technology Stack.....	52
7.2 Data Management Strategy.....	53

7.3 Scalability Architecture.....	54
8. Kesimpulan.....	54
9. Daftar Pustaka.....	55

Daftar Tabel

Tabel 2.1 Ubiquitous Language Reservation Context.....	6
Tabel 2.2 Ubiquitous Language Pricing Context.....	8
Tabel 2.3 Ubiquitous Language Inventory Context.....	9
Tabel 2.4 Ubiquitous Language Guest Context.....	10
Tabel 2.5 Ubiquitous Language Operations Context.....	11
Tabel 2.6 Ubiquitous Language Payment Context.....	12
Tabel 3.2 Dampak Domain Events Reservation Context Terhadap Konteks Lain.....	26
Tabel 3.3 Context Map Summary.....	27
Tabel 4.3 Reservation Context vs Pricing Context.....	29
Tabel 4.3.1.1 Reservation Context vs Pricing Context.....	29
Tabel 4.3.1.2 Reservation Context vs Inventory Context.....	30

Daftar Gambar

Gambar 3.1 Diagram Context Map.....	14
-------------------------------------	----

1. PENDAHULUAN

1.1 Tujuan Dokumen

Dokumen ini bertujuan untuk:

- Mendefinisikan Bounded Context untuk Sistem Reservasi Hotel
- Menyajikan Context Map yang menggambarkan hubungan antar bounded context
- Memilih dan mendesain detail satu konteks inti
- Memberikan panduan implementasi berdasarkan prinsip Domain-Driven Design (DDD)

1.2 Dasar Analisis

Dokumen ini disusun berdasarkan analisis domain sebelumnya yang mengidentifikasi:

- **2 Core Subdomain:** Booking & Reservation Management, Dynamic Pricing & Revenue Management
- **5 Supporting Subdomain:** Room Management, Guest Profile Management, Check-In/Out Operations, Housekeeping Management, Additional Services Management
- **4 Strategic Subdomain:** Authentication & Authorization, Payment Processing, Notification Services, Reporting & Analytics

2. IDENTIFIKASI OF BOUNDED CONTEXTS

Berdasarkan analisis subdomain, Sistem Reservasi Hotel dibagi menjadi **6 Bounded Context** utama:

2.1 Reservation Context [CORE]

Deskripsi:

Context yang mengelola seluruh lifecycle pemesanan kamar hotel dari pencarian hingga pembatalan.

Tanggung Jawab:

- Mengelola seluruh siklus pemesanan kamar (reservation lifecycle)
- Search dan availability checking real-time
- Pembuatan reservasi baru dengan validasi business rules
- Modifikasi reservasi (perubahan tanggal, tipe kamar, jumlah tamu)
- Pembatalan reservasi dengan perhitungan refund
- Waitlist management untuk kamar yang fully booked
- Overbooking strategy untuk optimasi occupancy
- Koordinasi dengan context lain untuk proses end-to-end

Ubiquitous Language:

Tabel 2.1 Ubiquitous Language Reservation Context

Term	Definition
------	------------

Reservation	Pemesanan kamar oleh tamu dengan detail tanggal, tipe kamar, dan harga
ReservationStatus	Status pemesanan: Pending, Confirmed, Checked-in, Checked-out, Cancelled
Availability	Ketersediaan kamar pada periode tertentu
Waitlist	Daftar tunggu untuk kamar yang fully booked
CancellationPolicy	Kebijakan pembatalan (refundable/non-refundable, deadline, penalty)
MinimumStay	Durasi minimum menginap yang required
OverbookingLimit	Batas maksimal overbooking yang diizinkan
ConfirmationCode	Kode unik untuk identifikasi reservasi

Domain Events:

- **ReservationCreated** - Reservasi baru berhasil dibuat
- **ReservationModified** - Reservasi diubah (tanggal/kamar)
- **ReservationCancelled** - Reservasi dibatalkan
- **ReservationConfirmed** - Pembayaran berhasil, reservasi confirmed
- **GuestCheckedIn** - Tamu melakukan check-in
- **GuestCheckedOut** - Tamu melakukan check-out
- **AvailabilityChanged** - Ketersediaan kamar berubah
- **WaitlistCreated** - Tamu ditambahkan ke waitlist

Alasan CORE:

- **Jantung bisnis hotel** - Setiap transaksi dimulai dari reservasi
- **Kompleksitas tinggi** - Business rules kompleks, state management sophisticated
- **Direct revenue impact** - Conversion rate bergantung pada reservation experience
- **Competitive advantage** - Cara mengelola reservation adalah differensiator utama
- **Integration hub** - Berinteraksi dengan hampir semua context lain

2.2 Pricing Context [CORE]

Deskripsi:

Context yang mengelola strategi penetapan harga dinamis untuk optimasi revenue hotel.

Tanggung Jawab:

- Dynamic rate calculation berdasarkan demand dan occupancy
- Revenue optimization melalui yield management
- Seasonal dan event-based pricing

- Discount strategy management (early bird, last minute deals)
- Competitive pricing analysis dan monitoring
- Rate forecasting dan recommendation
- Package pricing untuk bundled services

Ubiquitous Language:

Tabel 2.2 Ubiquitous Language Pricing Context

Term	Definition
RateStrategy	Strategi penetapan harga (Dynamic, Fixed, Package)
DynamicRate	Harga yang disesuaikan berdasarkan demand real-time
SeasonalPricing	Harga berdasarkan musim/periode (high season, low season)
YieldManagement	Optimasi pendapatan per kamar tersedia
OccupancyRate	Tingkat okupansi hotel (% kamar terisi)
ADR	Average Daily Rate - rata-rata harga harian
RevPAR	Revenue Per Available Room - metrik profitabilitas
DiscountRule	Aturan diskon (persentase, minimum stay, blackout dates)
CompetitorRate	Harga kompetitor untuk benchmarking

Domain Events:

- **RateCalculated** - Harga berhasil dihitung untuk permintaan
- **PriceAdjusted** - Harga disesuaikan berdasarkan demand
- **DiscountApplied** - Diskon diterapkan ke reservasi
- **SeasonalRateActivated** - Pricing musiman mulai berlaku
- **RevenueForecastUpdated** - Proyeksi revenue diperbarui

Alasan CORE:

- **Kunci profitabilitas** - Pricing strategy langsung impact revenue
 - **Intellectual property** - Algorithm pricing adalah competitive secret
 - **Kompleksitas sangat tinggi** - Multiple variables, mathematical modeling
 - **Strategic asset** - Continuous improvement untuk market advantage
 - **Data-driven decisions** - Requires sophisticated analytics
-

2.3 Inventory Context [SUPPORTING]

Deskripsi:

Konteks yang mengelola inventory kamar dan status operasional hotel.

Tanggung Jawab:

- Room inventory management (database seluruh kamar)
- Real-time status tracking kamar
- Room assignment ke reservasi
- Maintenance scheduling dan tracking
- Room blocking untuk maintenance atau events
- Room type management (Standard, Deluxe, Suite, dll)

Ubiquitous Language:

Tabel 2.3 Ubiquitous Language Inventory Context

Term	Definition
RoomInventory	Daftar lengkap semua kamar hotel dengan detail
RoomType	Tipe/kategori kamar (Standard, Deluxe, Suite)
RoomStatus	Status kamar: Available, Occupied, Cleaning, Maintenance, OutOfOrder
RoomAssignment	Penugasan nomor kamar spesifik ke reservasi
MaintenanceSchedule	Jadwal perawatan rutin atau perbaikan kamar
RoomBlocking	Blocking kamar untuk keperluan tertentu (event, maintenance)
RoomAmenities	Fasilitas dalam kamar (TV, AC, minibar, dll)

Domain Events:

- **RoomStatusChanged** - Status kamar berubah
- **RoomAssigned** - Kamar ditugaskan ke reservasi
- **RoomBlocked** - Kamar di-block untuk maintenance/event
- **MaintenanceScheduled** - Maintenance dijadwalkan
- **MaintenanceCompleted** - Maintenance selesai

Alasan SUPPORTING:

- Penting untuk operasional tapi tidak differentiating
 - Proses standar di industri hospitality
 - Best practices sudah well-established
-

2.4 Guest Context [SUPPORTING]

Deskripsi:

Context yang mengelola profil tamu dan program loyalitas.

Tanggung Jawab:

- Guest profile management (data pribadi, kontak)
- Preference tracking (tipe kamar, lantai, dietary restrictions)
- Stay history tracking
- Loyalty program management
- Membership tier management (Silver, Gold, Platinum)
- Guest authentication dan authorization

Ubiquitous Language:

Tabel 2.4 Ubiquitous Language Guest Context

Term	Definition
GuestProfile	Profil lengkap tamu dengan informasi pribadi
Preference	Preferensi tamu (room type, floor, smoking/non-smoking)
StayHistory	Riwayat menginap tamu di hotel
LoyaltyMember	Anggota program loyalitas hotel
MembershipTier	Tingkatan membership: Silver, Gold, Platinum
LoyaltyPoints	Poin yang dikumpulkan dari menginap
PointsRedemption	Penukaran poin untuk reward

Domain Events:

- **GuestRegistered** - Tamu baru mendaftar
- **ProfileUpdated** - Profil tamu diperbarui
- **LoyaltyPointsEarned** - Poin earned dari stay
- **LoyaltyPointsRedeemed** - Poin ditukar untuk reward
- **TierUpgraded** - Tier membership naik

Alasan SUPPORTING:

- Mendukung personalization tapi bukan core differentiator
 - CRM system standar bisa diadaptasi
 - Data management relatif straightforward
-

2.5 Operations Context [SUPPORTING]

Deskripsi:

Context yang mengelola operasional harian hotel.

Tanggung Jawab:

- Check-in operations (guest verification, key issuance)
- Check-out operations (bill settlement, key return)
- Housekeeping task management
- Cleaning schedule dan assignment
- Additional services management (room service, laundry)
- Facility reservation (restaurant, spa, meeting rooms)

Ubiquitous Language:

Tabel 2.5 Ubiquitous Language Operations Context

Term	Definition
CheckInProcess	Proses kedatangan dan registrasi tamu
CheckOutProcess	Proses keberangkatan dan settlement bill
CleaningTask	Tugas pembersihan kamar
HousekeepingStaff	Staff kebersihan yang assigned
ServiceRequest	Permintaan layanan tambahan dari tamu
FacilityReservation	Pemesanan fasilitas hotel (spa, restaurant)
RoomServiceOrder	Pesanan room service

Domain Events:

- **CheckInCompleted** - Check-in selesai
- **CheckOutCompleted** - Check-out selesai
- **CleaningTaskAssigned** - Task assigned ke staff
- **CleaningCompleted** - Pembersihan selesai
- **ServiceRequestCreated** - Request layanan dibuat
- **ServiceRequestFulfilled** - Request dipenuhi

Alasan SUPPORTING:

- Proses operasional standar industri
- Tidak memberikan competitive advantage
- Best practices well-documented

2.6 Payment Context [STRATEGIC]

Deskripsi:

Context yang mengelola proses pembayaran dan transaksi keuangan.

Tanggung Jawab:

- Payment processing (credit card, bank transfer, digital wallet)
- Payment gateway integration (Midtrans, Stripe, Xendit)
- Transaction management dan tracking
- Deposit handling dan partial payment
- Refund processing untuk cancellation
- Invoice generation
- Payment reconciliation

Ubiquitous Language:

Tabel 2.6 Ubiquitous Language Payment Context

Term	Definition
Transaction	Transaksi pembayaran
PaymentMethod	Metode pembayaran (CreditCard, BankTransfer, Wallet)
PaymentStatus	Status pembayaran: Pending, Completed, Failed, Refunded
Invoice	Faktur pembayaran
Refund	Pengembalian dana
Deposit	Uang muka/deposit
PaymentGateway	Gateway pihak ketiga untuk processing

Domain Events:

- **PaymentInitiated** - Pembayaran dimulai
- **PaymentReceived** - Pembayaran berhasil
- **PaymentFailed** - Pembayaran gagal
- **RefundProcessed** - Refund berhasil diproses
- **InvoiceGenerated** - Invoice dibuat

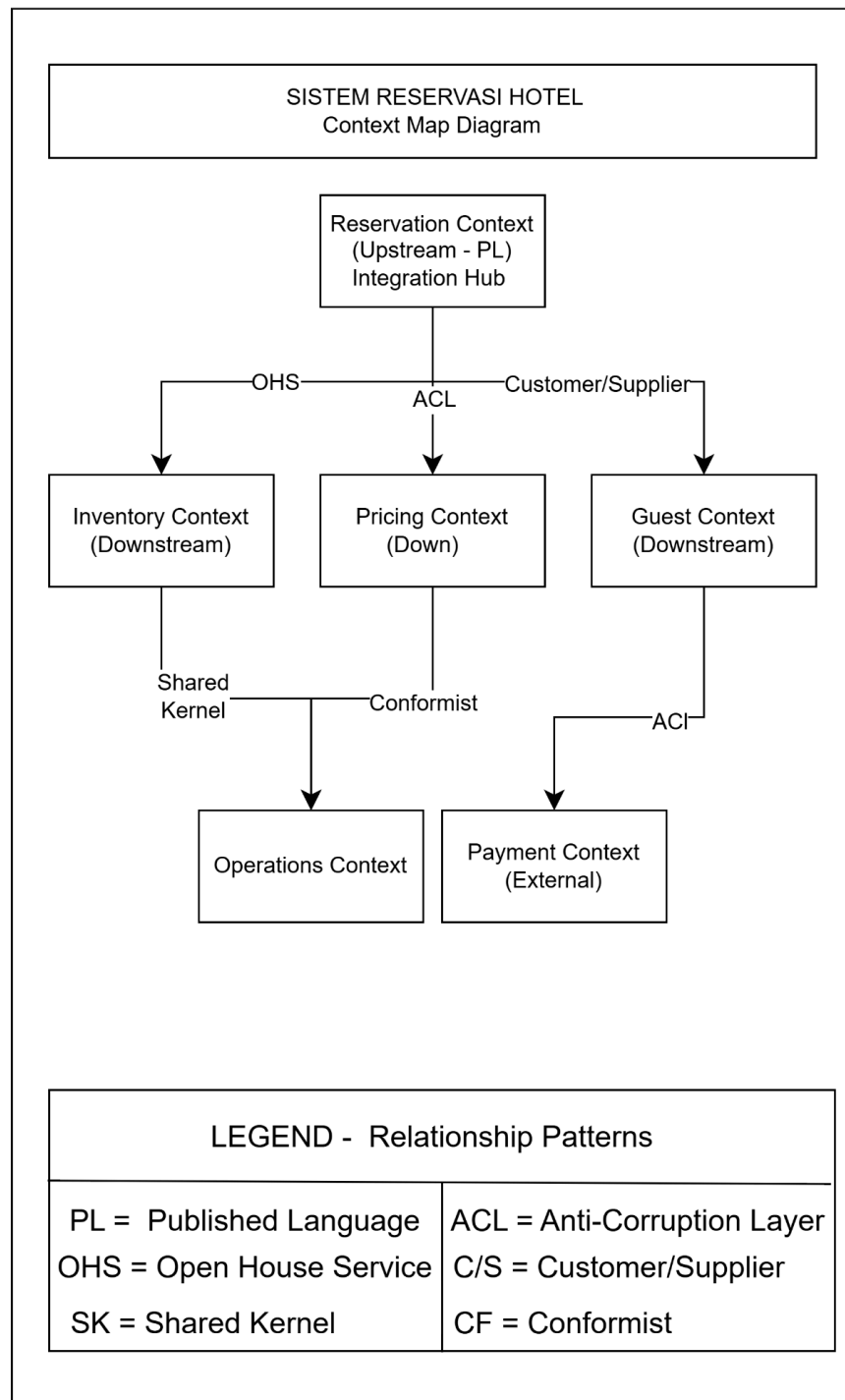
Alasan STRATEGIC:

- Commodity service dengan mature solutions
- Security compliance (PCI-DSS) complex

- Better use third-party specialized services
- Cost-effective integration approach

3. CONTEXT MAP

3.1 Diagram Context Map



Gambar 3.1 Diagram Context Map

Stateless Application Servers:

- No session state in app servers
- All state in database or Redis
- Enable horizontal scaling tanpa sticky sessions
- Auto-scaling based on load

Multi-Level Caching Strategy:

Level 1: Application Cache (In-Memory)

- Very frequently accessed data
- Latency: <1ms
- Size: Limited (100MB per instance)
- Example: Configuration, static data

Level 2: Redis Cache (Distributed)

- Shared across all app servers
- Latency: ~5ms
- Size: Larger capacity (10GB+)
- Example: Availability data, pricing

Level 3: Database (Source of Truth)

- Persistent storage
- Latency: ~50ms
- Unlimited capacity
- All data eventually here

Cache Invalidation:

- Event-driven invalidation
- TTL-based expiration
- Cache-aside pattern
- Write-through untuk critical data

Concurrency Control Mechanisms:

1. Optimistic Locking:

- Version column dalam aggregates
- Check version pada update
- Retry pada conflict
- Good for low-contention scenarios

2. Distributed Locks (Redis):

- Lock acquisition dengan timeout
- Automatic release on expiration
- Used for availability updates
- Prevents double-reservation

3. Pessimistic Locking (Database):

- SELECT FOR UPDATE
- Used sparingly untuk critical sections
- Short lock duration
- Avoid deadlocks

3.2 Relationship Patterns Detail

3.2.1 Reservation dengan Inventory (OHS/PL)

Pattern: Open Host Service & Published Language

Direction: Reservation (Upstream) mengarah ke Inventory (Downstream)

Karakteristik:

- Reservation mendefinisikan protokol standar untuk pengecekan ketersediaan
- Inventory menyediakan REST API yang terdokumentasi dengan baik
- Multiple contexts dapat mengkonsumsi data inventory dengan interface yang sama
- Published Language: JSON schema untuk data kamar dan ketersediaan

Alasan Pemilihan:

- Inventory adalah foundational service yang digunakan banyak context
- Standardisasi API mengurangi coupling
- Dokumentasi yang jelas untuk consumers
- Versioning API untuk backward compatibility

Integration Method:

- REST API dengan JSON schema
- Endpoint GET untuk mendapatkan status kamar berdasarkan roomTypeId
- Endpoint GET untuk cek ketersediaan berdasarkan rentang tanggal (contoh: tanggal 1 Maret 2025 sampai 5 Maret 2025)

Benefits:

- **Loose coupling** - Inventory dapat evolve secara independen
 - **Reusability** - API dapat digunakan contexts lain
 - **Clear contract** - Published Language sebagai dokumentasi
-

3.2.2 Reservation dengan Pricing (ACL)

Pattern: Anti-Corruption Layer

Direction: Reservation (Upstream) mengarah ke Pricing (Downstream)

Karakteristik:

- Pricing Context memiliki model yang sangat kompleks
- Reservation tidak ingin terikat dengan kompleksitas pricing algorithm
- ACL mentranslasi complex pricing model ke simple price value
- ACL melindungi Reservation dari perubahan pricing strategy

Alasan Pemilihan:

- Pricing algorithm sering berubah (weekly/monthly adjustments)
- Reservation hanya butuh final price, tidak perlu tahu calculation details
- Isolasi domain model Reservation dari pricing complexity
- Flexibility untuk mengganti pricing engine tanpa impact Reservation
- Pricing mungkin bergantung pada ML models yang kompleks

Integration Method:

ACL Layer - PricingAdapter:

Interface yang dilihat Reservation Context:

- get_room_price(room_type, date_range, guest_profile) → Money

Input sederhana:

- room_type: "DELUXE"
- date_range: { start: "2025-03-01", end: "2025-03-05" }
- guest_profile: { loyaltyTier: "GOLD" }

Output sederhana:

- Money: { amount: 5000000, currency: "IDR" }

- ACL Layer menyediakan simple interface
- Input: room type, date range, guest profile
- Output: simple Money value
- ACL handle semua kompleksitas internal Pricing

Benefits:

- **Protection** - Domain model tidak tercemar pricing complexity
- **Flexibility** - Pricing dapat change algorithm tanpa break Reservation
- **Simplicity** - Reservation hanya deal dengan simple price value
- **Testability** - ACL dapat di-mock untuk testing
- **Evolution** - Both contexts dapat evolve independently

Trade-offs:

- Additional layer means slight performance overhead
 - ACL code perlu maintained
 - Need clear ownership (biasanya Reservation team owns the ACL)
-

3.2.3 Reservation dengan Guest (Customer/Supplier)

Pattern: Customer/Supplier

Direction: Bidirectional

Karakteristik:

- Guest Context bertindak sebagai **Supplier** - menyediakan guest information
- Reservation Context bertindak sebagai **Customer** - mengkonsumsi guest data
- **Reverse direction:** Reservation provide reservation history back to Guest Context
- Mutual benefit relationship dengan clear boundaries

Alasan Pemilihan:

- Guest dan Reservation saling membutuhkan data satu sama lain
- Neither context dominates the relationship
- Clear supplier-customer dynamic untuk each direction
- Balanced power relationship

Integration Method:

Direction 1: Guest (Supplier) → Reservation (Customer)

Reservation queries guest profile dan preferences:

- GET /api/guest/v1/profile/{guestId}

Response: {

guestId: "GUEST-12345",

name: "John Doe",

```
email: "john@example.com",
phone: "+62812345678",
preferences: {
  roomType: "DELUXE",
  floor: "HIGH",
  smoking: false,
  bedType: "KING"
},
loyaltyTier: "GOLD",
loyaltyPoints: 5000
}
```

- GET /api/guest/v1/{guestId}/preferences

Response: { preferredRoomTypes: [...], specialRequests: [...] }

Direction 2: Reservation (Supplier) → Guest (Customer)

Guest context enriched dengan reservation history:

- POST /api/guest/v1/{guestId}/reservation -history

```
Request: {
  reservationId: "RES-98765",
  hotelId: "HOTEL-001",
  checkInDate: "2025-03-01",
  checkOutDate: "2025-03-05",
  roomType: "DELUXE",
  totalAmount: 5000000,
}
```

```
status: "COMPLETED"

}
```

- POST /api/guest/v1/{guestId}/loyalty-points

```
Request: {

  reservationId: "RES-98765",

  points: 500,

  reason: "STAY_COMPLETED"

}
```

Benefits:

- **Reciprocal relationship** - Both contexts benefit
- **Clear ownership** - Each context owns its core data
- **Loose coupling** - Communication via well-defined APIs
- **Flexibility** - Each can evolve independently

Coordination Requirements:

- Regular sync meetings antara teams
 - API contract versioning strategy
 - Breaking change notifications
 - Depreciation policy untuk old APIs
-

3.2.4 Pricing dengan Operations (Conformist)

Pattern: Conformist

Direction: Pricing (Upstream) mengarah ke Operations (Downstream)

Karakteristik:

- Operations Context **conforms** sepenuhnya ke Pricing model
- Operations tidak memiliki power untuk influence pricing decisions
- Operations accept pricing data as-is tanpa translation
- No negotiation atau customization dari Operations side

Alasan Pemilihan:

- Operations hanya need read-only access ke pricing data
- Pricing strategy adalah domain dari Pricing Context

- Operations tidak butuh kompleksitas ACL karena hanya display prices
- Cost-effective - no need untuk translation layer

Integration Method:

Operations directly consume Pricing API:

- GET /api/pricing/v1/rates?date=2025-03-01&roomType=DELUXE_001

Response: {

date: "2025-03-01",

roomType: "DELUXE_001",

baseRate: 1200000,

seasonalAdjustment: 1.2,

finalRate: 1440000,

currency: "IDR",

discountsAvailable: [

{ type: "EARLY_BIRD", discount: 10, minDaysAdvance: 30 },

{ type: "WEEKEND", discount: 0 }

]

}

Operations use this untuk:

- Display pada front desk system
- Show kepada guest saat check-in
- Print on invoices
- No transformation, just display as-is

Benefits:

- **Simplicity** - No translation layer needed
- **Consistency** - Pricing displayed sama di semua contexts
- **Efficiency** - Direct API consumption
- **Clarity** - Single source of truth untuk pricing

Trade-offs:

- Operations vulnerable to Pricing API changes
- Acceptable karena Operations bukan critical business logic
- Pricing Context maintain backward compatibility

Risk Mitigation:

- Pricing provides stable API with versioning
 - Breaking changes communicated well in advance
 - Operations dapat cache pricing data untuk resilience
 - SLA agreement untuk API availability
-

3.2.5 Guest dengan Payment (ACL)

Pattern: Anti-Corruption Layer

Direction: Payment (External) mengarah ke Guest (Internal)

Karakteristik:

- Payment Context adalah **external third-party** service
- ACL protects internal domain dari external payment gateway model
- Security isolation untuk sensitive payment data
- Translation payment events ke domain events

Alasan Pemilihan:

- Payment gateway (Midtrans, Stripe) adalah external system
- Protect internal model dari vendor-specific implementation
- Security compliance - isolate PCI-DSS data
- Flexibility untuk change payment provider tanpa impact domain

Integration Method:

ACL Layer - PaymentGatewayAdapter:

Domain Interface (yang dilihat internal contexts):

- initiate_payment(reservation_id, amount, guest_id) → PaymentInitiationResult

Input domain terms:

- reservation_id: "RES-12345"

- amount: Money(5000000, "IDR")
- guest_id: "GUEST-67890"

Output domain terms:

- PaymentInitiationResult: {
 payment_url: "https://payment.hotel.com/pay/xyz",
 payment_token: "TOKEN-ABC123",
 expires_at: "2025-03-01T10:00:00Z"
}
- process_refund(reservation_id, amount) → RefundResult
- check_payment_status(payment_token) → PaymentStatus

Internal ACL transformation:

Untuk Midtrans:

1. Transform domain PaymentRequest ke Midtrans format:

```
{  
  transaction_details: {  
    order_id: "RES-12345",  
    gross_amount: 5000000  
  },  
  customer_details: {  
    customer_id: "GUEST-67890",  
    email: "...",  
    phone: "..."  
  },  
}
```

```
item_details: [...]  
}
```

2. Call Midtrans API: POST /v2/charge

3. Transform Midtrans response ke domain model:

Extract: redirect_url, token, expiry_time

4. Publish domain event (bukan gateway event):

PaymentInitiated(reservation_id, amount, timestamp)

Untuk Stripe (jika switch provider):

ACL transform ke Stripe format - internal contexts tidak affected

Benefits:

- **Vendor independence** - Easy to switch payment providers
 - **Security isolation** - Payment details tidak leak ke domain
 - **Domain purity** - Domain model tidak tercemar external concepts
 - **Testability** - Mock ACL untuk testing tanpa real payment
-

3.2.6 Inventory dengan Operations (Shared Kernel)

Pattern: Shared Kernel

Direction: Bidirectional tight coupling

Karakteristik:

- Inventory dan Operations share subset of domain model
- Common model untuk RoomStatus dan RoomAssignment
- Both contexts need real-time synchronization
- Shared responsibility untuk data consistency

Alasan Pemilihan:

- RoomStatus updates dari Operations impact Inventory directly
- Real-time requirements memerlukan tight integration
- Cost of synchronization overhead too high dengan loose coupling

- Both teams coordinate untuk maintain shared model

Shared Elements:

1. RoomStatus Enumeration:

- AVAILABLE: Kamar siap untuk ditempati
- OCCUPIED: Kamar sedang ditempati guest
- CLEANING: Kamar sedang dibersihkan housekeeping
- MAINTENANCE: Kamar dalam perbaikan
- OUT_OF_ORDER: Kamar tidak dapat digunakan
- BLOCKED: Kamar di-block untuk alasan tertentu

2. RoomAssignment:

- room_number: string
- reservation_id: string
- guest_id: string
- status: RoomStatus
- assigned_at: datetime
- expected_checkout: datetime

3. Status Transition Rules:

- AVAILABLE → OCCUPIED (saat check-in)
- OCCUPIED → CLEANING (saat check-out)
- CLEANING → AVAILABLE (setelah cleaning done)
- ANY → MAINTENANCE (saat ada issue)
- MAINTENANCE → AVAILABLE (setelah fixed)

4. Shared Business Rules:

- Status transitions yang valid
- Assignment constraints
- Cleaning time expectations (30 min for standard room)

Integration Method:

Shared Database Table:

Table: room_status (shared by both contexts)

Columns:

- room_number (PK)
- current_status
- reservation_id (FK, nullable)

- guest_id (FK, nullable)
- status_changed_at
- changed_by
- version (untuk optimistic locking)

Both contexts dapat:

- Read current room status
- Update room status
- Query available rooms
- Assign rooms to reservations

Coordination Protocol:

1. Operations updates status saat check-in/check-out
2. Inventory reflects status changes immediately
3. Housekeeping (via Operations) updates saat cleaning
4. Inventory sees real-time cleaning status
5. Both contexts respect shared transition rules

Benefits:

- **Performance** - No API overhead
- **Consistency** - Real-time data sharing
- **Simplicity** - No translation layer

Trade-offs:

- **Tight coupling** - Changes require coordination antara teams
- **Risk** - One context can potentially break the other
- **Database dependency** - Both contexts share same database
- **Accepted karena:**
 - Real-time requirements justify the coupling
 - Both contexts have equal ownership

- Clear governance untuk shared model changes
- Performance benefits outweigh risks

Governance:

- **Joint ownership** - Both teams own the shared kernel
- **Change process:**
 1. Proposal dari either team
 2. Review bersama
 3. Impact analysis
 4. Coordinated deployment
- **Versioning strategy** untuk shared schema
- **Regular sync meetings** untuk alignment
- **Documentation** untuk shared model maintained jointly

Risk Mitigation:

- Optimistic locking untuk prevent conflicts
- Comprehensive testing untuk shared logic
- Rollback procedures clearly defined
- Monitoring untuk shared kernel health
- Escape hatch: Can migrate to event-driven jika coupling jadi masalah

Tabel 3.2 Dampak Domain Events Reservation Context Terhadap Konteks Lain

Domain Event	Deskripsi Singkat	Context yang Menerima Dampak	Alasan Dependensi
ReservationCreated	Reservasi baru dibuat	Inventory, Pricing, Guest, Payment	Inventory mengurangi ketersediaan; Pricing mencatat demand; Guest menambah riwayat; Payment memulai proses pembayaran

ReservationModified	Perubahan tanggal, kamar, tamu	Inventory, Pricing, Guest	Availability perlu di-adjust; Pricing recalculates; Guest update stay details
ReservationCancelled	Pembatalan reservasi	Inventory, Payment, Guest	Inventory melepas kamar; Payment memproses refund; Guest update riwayat
ReservationConfirmed	Pembayaran sukses	Operations, Guest	Operations menyiapkan check-in; Guest dapat loyalty point
GuestCheckedIn	Tamu check-in	Operations, Inventory	Update status kamar menjadi occupied
GuestCheckedOut	Tamu check-out	Operations, Inventory, Guest	Room menjadi dirty/cleaning; Guest update stay history
AvailabilityChanged	Perubahan ketersediaan kamar	Reservation, Pricing	Reservation dapat update availability; Pricing update forecast
WaitlistCreated	Tamu masuk waitlist	Notification, Reservation	Notifikasi untuk follow-up; Reservation integrasi dengan antrian kamar

3.3 Context Map Summary Table

Tabel 3.3 Context Map Summary

Upstream Context	Downstream Context	Pattern	Rationale	Integration
Reservation	Inventory	OHS/PL	Standardized availability API	REST API

Reservation	Pricing	ACL	Protect from pricing complexity	ACL Adapter
Reservation	Guest	C/S	Bidirectional data exchange	REST API
Pricing	Operations	Conformist	Operations conform to pricing	REST API
Payment (External)	Guest	ACL	Protect from external gateway	ACL Adapter
Inventory	Operations	Shared Kernel	Real-time status sync	Shared DB

Pattern Selection Criteria:

OHS/PL - Pilih ketika:

- Service digunakan oleh multiple contexts
- Need standardized interface
- Want to reduce coupling
- Clear documentation important

ACL - Pilih ketika:

- External system atau complex model
- Need protect domain purity
- Frequent changes expected di upstream
- Want flexibility untuk switch providers

Customer/Supplier - Pilih ketika:

- Bidirectional dependency
- Neither dominates relationship
- Mutual benefit dari collaboration
- Clear negotiable interfaces

Conformist - Pilih ketika:

- Downstream tidak perlu customize
- Upstream model acceptable
- Cost of ACL tidak justified
- Read-only atau display purposes

Shared Kernel - Pilih ketika:

- Real-time requirements critical
- Both teams closely collaborate
- Performance overhead dari loose coupling too high

- Clear governance dapat diestablish

4. PEMILIHAN KONTEKS INTI

4.1 Konteks Inti yang Dipilih: RESERVATION CONTEXT [CORE]

4.2 Kriteria Evaluasi

Untuk memilih konteks inti, dilakukan evaluasi berdasarkan kriteria berikut:

Metodologi Evaluasi:

- Setiap kriteria diberi bobot sesuai importance untuk business
- Setiap context di-score 1-10 untuk tiap kriteria
- Total score = Σ (Kriteria Score \times Bobot)

4.3 Analisis Komparatif

Reservation Context vs Pricing Context

Tabel 4.3 Reservation Context vs Pricing Context

Kriteria	Bobot	Reservation	Pricing	Inventory	Guest	Operations	Payment
Business Value	30%	10	10	7	6	5	3
Complexity	25%	10	9	6	5	4	3
Integration Needs	20%	10	5	7	6	8	4
User Impact	15%	10	3	5	7	8	6
Strategic Importance	10%	10	10	5	5	4	2
Total Score	100%	10.0	7.8	6.3	5.9	5.7	3.5

4.3.1 Analisis Komparatif

Reservation Context vs Pricing Context

Tabel 4.3.1.1 Reservation Context vs Pricing Context

Aspek	Reservation Context (Konteks Pemesanan)	Pricing Context (Konteks Penetapan Harga)
-------	--	--

Business Impact	Direct customer touchpoint, conversion driver	Indirect, backend optimization
Complexity	Workflow complexity, state management	Algorithm complexity, mathematical
Integration	Hub - interacts with ALL contexts	Limited - mainly Reservation
User Facing	Yes - critical UX	No - backend only
Transaction Volume	Very High (every reservation)	Medium (calculation on-demand)
Real-time Requirement	Critical (availability must be accurate)	Moderate (rates can be cached)
Development Priority	Phase 1 - Foundation	Phase 2 - Optimization

Reservation Context vs Inventory Context

Tabel 4.3.1.2 Reservation Context vs Inventory Context

Aspek	Reservation Context (Konteks Pemesanan)	Inventory Context (Konteks Inventaris)
Business Logic Complexity	State machine complex, business rules, concurrency handling, payment integration, cancellation policies	CRUD operations, simple counting, maintenance scheduling
Strategic Value	Competitive differentiator, direct revenue generator	Foundational service, necessary but not differentiating
Change Frequency	Frequently changes - business requirements evolve, promo strategies change, integration with external systems	Stable - room inventory rarely changes
Team Investment	Senior developers, continuous iteration, intensive testing	Junior developers, predictable maintenance

4.4 Justifikasi Final

Reservation Context dipilih sebagai Core Domain karena:

Alasan Bisnis:

- Menentukan success atau failure dari reservation conversion
- Direct impact ke revenue dan customer satisfaction

- Kompetitor differentiation melalui reservation experience
- Highest ROI dari investment development

Alasan Teknis:

- Paling complex state management dan business rules
- Memerlukan real-time coordination dengan multiple contexts
- Concurrency control critical untuk inventory accuracy
- Integration hub untuk seluruh system

Alasan Strategis:

- Continuous innovation diperlukan untuk stay competitive
- User expectations terus evolve
- Foundation untuk future features seperti dynamic pricing, personalization, loyalty programs

Konsekuensi Keputusan:

- Best developers assigned ke Reservation team
- Highest test coverage requirement
- More frequent iterations dan releases
- Significant architecture investment
- Domain expertise crucial untuk team members

Prioritas Implementasi (Reservation vs Pricing): Meskipun *Pricing Context* juga merupakan Core Domain yang krusial untuk profitabilitas, *Reservation Context* diprioritaskan sebagai fokus utama pengembangan fase awal (M02-M04). Hal ini dikarenakan *Reservation Context* merupakan **tulang punggung operasional (operational backbone)**; tanpa adanya sistem reservasi yang berfungsi untuk mencatat transaksi dan mengelola inventaris, strategi *dynamic pricing* yang canggih tidak dapat dieksekusi. Oleh karena itu, *Reservation* harus matang terlebih dahulu sebagai fondasi backend sebelum fitur optimasi seperti *Pricing* dikembangkan.

5. STRATEGIC DESIGN - RESERVATION CONTEXT

5.1 Core Domain Model Overview

5.1.1 Aggregates

1. Reservation Aggregate (Root)

Purpose: Mengelola complete lifecycle sebuah pemesanan kamar dari creation hingga completion atau cancellation.

Core Responsibilities:

- Enforce reservation business rules
- Manage state transitions dengan validation

- Calculate refunds sesuai cancellation policy
- Coordinate dengan external contexts via events
- Maintain consistency boundary untuk reservation data

Key Attributes:

Identity & References:

- `reservation_id`: UUID (primary identifier, immutable)
- `guest_id`: UUID (reference ke Guest Context)
- `room_type_id`: String (reference ke Inventory Context)

Reservation Details:

- `date_range`: DateRange value object (check-in, check-out dates)
- `guest_count`: GuestCount value object (adults, children)
- `total_amount`: Money value object (price dengan currency)
- `cancellation_policy`: CancellationPolicy value object

Status & State:

- `status`: ReservationStatus enum (current state)
- `reservation_source`: ReservationSource enum (channel asal reservation)
- `confirmation_code`: String (unique code untuk guest reference)

Collections:

- `special_requests`: List[SpecialRequest] (child entities dalam aggregate)
- Request type (early check-in, extra bed, etc.)
- Description
- Fulfillment status

Metadata:

- `created_at`: Timestamp
- `modified_at`: Timestamp
- `created_by`: String (user/system yang create)
- `version`: Integer (untuk optimistic locking)

Key Invariants (Must Always Be True):

1. Check-out date MUST be after check-in date
2. Status transitions MUST follow valid state machine
3. Guest count MUST NOT exceed room type capacity
4. Total amount MUST be positive
5. Minimum stay requirements MUST be met
6. Cancellation refund MUST follow policy rules
7. Only PENDING or CONFIRMED reservations can be modified
8. CHECKED_OUT reservations CANNOT be deleted (audit trail)

Core Methods (Behavior):**Creation & Modification:**

- ``create()``: Initialize new reservation dengan validation
- ``modify()``: Update details (dates, room type, guest count)
- ``add_special_request()``: Add guest request
- ``update_amount()``: Recalculate price after changes

State Transitions:

- ``confirm()``: Mark as confirmed after payment
- ``check_in()``: Mark guest as checked in
- ``check_out()``: Complete stay
- ``cancel()``: Cancel dengan refund calculation
- ``mark_no_show()``: Guest didn't arrive

Queries:

- ``is_modifiable()``: Check if can be modified
- ``is_cancellable()``: Check if can be cancelled
- ``calculate_refund()``: Calculate refund amount

- ``get_nights()``: Number of nights staying

2. Availability Aggregate (Root)

Purpose: Track dan manage ketersediaan kamar untuk specific date dan room type.

Design Rationale:

- **Separate dari Reservation** untuk avoid contention
- **Granular per (RoomType, Date)** untuk better concurrency
- **Eventual consistency** dengan Reservation acceptable

Core Responsibilities:

- Maintain accurate room counts
- Handle reservations dan releases
- Enforce overbooking limits
- Support concurrent updates safely

Key Attributes:

Composite Identity:

- ``room_type_id``: String (part of identity)
- ``date``: Date (part of identity)
- Composite key: (room_type_id, date) uniquely identifies aggregate

Capacity Tracking:

- ``total_rooms``: Integer (total kamar dari room type ini)
- ``reserved_rooms``: Integer (kamar yang sudah di-reserve)
- ``blocked_rooms``: Integer (kamar yang di-block untuk maintenance/event)

Configuration:

- ``overbooking_threshold``: Integer (max overbooking allowed)

Computed Properties (Not Stored):

- ``available_rooms``: $\text{total_rooms} - \text{reserved_rooms} - \text{blocked_rooms}$
- ``is_fully_booked``: $\text{available_rooms} \leq 0$
- ``can_overbook``: $\text{available_rooms} < 0$ but within threshold

Key Invariants:

1. ``total_rooms`` ≥ 0
2. ``reserved_rooms`` ≥ 0
3. ``blocked_rooms`` ≥ 0
4. ``reserved_rooms`` + ``blocked_rooms`` \leq ``total_rooms`` + ``overbooking_threshold``
5. If blocking rooms, must provide reason

Core Methods:

Availability Operations:

- ``check_availability(count)``: Check if count rooms available
- ``reserve_rooms(count)``: Decrease available, increase reserved
- ``release_rooms(count)``: Increase available, decrease reserved
- ``block_rooms(count, reason)``: Block for maintenance/events
- ``unblock_rooms(count)``: Unblock after maintenance

Query Methods:

- ``get_available_count()``: Current available rooms
- ``can_accommodate(count)``: Check if can fit count guests
- ``is_overbooked()``: Check if currently overbooked

Concurrency Handling:

- Optimistic locking via version field
- Compensating transaction if overbooking detected
- Event published on availability changes

3. Waitlist Aggregate (Root)

Purpose: Manage waiting list for fully booked dates, dengan priority handling dan notification.

Design Rationale:

- **Separate aggregate** karena independent lifecycle
- **Can exist without Reservation** (not yet converted)
- **Different business rules** dari normal reservation

Core Responsibilities:

- Track guests waiting for availability
- Manage priority ordering (VIP guests first)
- Handle expiry (waitlist entries expire after X days)
- Support conversion to actual reservation

Key Attributes:

Identity:

- ``waitlist_id``: UUID (primary identifier)

Request Details:

- ``guest_id``: UUID (which guest waiting)
- ``room_type_id``: String (desired room type)
- ``requested_dates``: DateRange (when they want to stay)
- ``guest_count``: GuestCount (number of guests)

Status & Priority:

- ``priority``: Priority enum (NORMAL, HIGH, VIP)
- ``status``: WaitlistStatus enum (ACTIVE, CONVERTED, EXPIRED)

Timestamps:

- ``created_at``: When added to waitlist
- ``expires_at``: When entry expires (typically 7-14 days)
- ``notified_at``: When guest was last notified

Key Invariants:

1. Only ACTIVE entries can be converted
2. Expired entries cannot be processed
3. Priority must be valid enum value
4. Expiry date must be after creation date
5. Converted entries must reference reservation

Core Methods:

Lifecycle:

- ``add_to_waitlist()``: Create entry dengan priority
- ``convert_to_reservation(reservation_id)``: Mark as converted
- ``expire()``: Mark as expired
- ``extend_expiry()``: Extend expiry date

Priority Management:

- ``upgrade_priority()``: Upgrade to higher priority
- ``calculate_priority_score()``: For ordering waitlist

Notification:

- ``mark_notified()``: Record notification sent
- ``should_notify_again()``: Check if time for reminder

5.1.2 Value Objects

1. DateRange

Purpose: Encapsulate check-in to check-out period dengan validation dan utility methods.

Attributes:

- `start`: Date (check-in date)
- `end`: Date (check-out date)

Characteristics:

- Immutable (frozen)
- Self-validating
- Equals based on values

Validation Rules:

- End must be after start
- Start cannot be in the past
- Maximum stay limit (e.g., 30 nights)

Utility Methods:

- `nights()`: Calculate number of nights
- `overlaps(other)`: Check if overlaps with another range
- `contains(date)`: Check if date within range
- `extend(days)`: Create new range extended by days
- `shift(days)`: Create new range shifted by days

Business Logic:

- Weekend detection (Friday-Sunday)
- Season detection (high season, low season)
- Holiday detection (special rates)

2. Money

Purpose: Represent monetary amounts dengan currency, preventing calculation errors.

Attributes:

- ``amount``: Decimal (precise decimal for currency)
- ``currency``: String (ISO code: IDR, USD, EUR)

Characteristics:

- Immutable
- Precise decimal arithmetic (no float errors)
- Currency-aware operations

Validation:

- Amount cannot be negative
- Currency must be supported

Operations:

- ``add(other)``: Add money (same currency only)
- ``subtract(other)``: Subtract money
- ``multiply(factor)``: Multiply by scalar
- ``percentage(percent)``: Calculate percentage
- ``split(parts)``: Split into equal parts
- ``convert_to(currency)``: Convert currency (with exchange rate)

3. ReservationStatus

Purpose: Represent lifecycle state dengan valid transitions.

States:

- ``PENDING``: Created, awaiting payment
- ``CONFIRMED``: Payment received
- ``CHECKED_IN``: Guest arrived
- ``CHECKED_OUT``: Stay completed
- ``CANCELLED``: Reservation cancelled
- ``NO_SHOW``: Guest didn't arrive

State Machine:

Valid Transitions:

- PENDING → CONFIRMED (after payment)
- PENDING → CANCELLED (cancelled before payment)
- CONFIRMED → CHECKED_IN (guest arrives)
- CONFIRMED → CANCELLED (cancelled after payment, with refund)
- CONFIRMED → NO_SHOW (didn't arrive)
- CHECKED_IN → CHECKED_OUT (normal checkout)

Invalid Transitions:

- CHECKED_OUT → any (terminal state)
- CANCELLED → any except audit changes (terminal state)
- NO_SHOW → any (terminal state)

4. CancellationPolicy

Purpose: Encapsulate refund calculation logic based on policy type.

Attributes:

- `policy_type`: PolicyType enum (FLEXIBLE, MODERATE, STRICT, NON_REFUNDABLE)
- `free_cancellation_hours`: Integer (hours before check-in for free cancellation)
- `penalty_percentage`: Decimal (penalty if cancelled after deadline)

Policy Types:

FLEXIBLE:

- Free cancellation up to 24 hours before check-in
- Full refund if cancelled within deadline
- No penalty

MODERATE:

- Free cancellation up to 7 days before check-in
- 50% refund if cancelled 3-7 days before
- No refund if cancelled <3 days before

STRICT:

- Free cancellation up to 14 days before check-in
- 25% refund if cancelled 7-14 days before
- No refund if cancelled <7 days before

NON_REFUNDABLE:

- No cancellation allowed
- No refund under any circumstances
- (Usually lowest price)

Refund Calculation Logic:

calculate_refund(total_amount, cancellation_date, check_in_date):

1. Calculate hours until check-in

2. If within free_cancellation_hours:

return total_amount (100% refund)

3. Else:

penalty = total_amount × penalty_percentage

refund = total_amount - penalty

return max(refund, 0)

5. SpecialRequest

Purpose: Track guest special requests dengan fulfillment status.

Attributes:

- `request_type`: RequestType enum
- `description`: String
- `fulfilled`: Boolean
- `notes`: String (optional)

Request Types:

- `EARLY_CHECKIN`: Request check-in earlier than standard time
- `LATE_CHECKOUT`: Request check-out later than standard time
- `EXTRA_BED`: Request additional bed in room
- `CRIB`: Request baby crib
- `HIGH_FLOOR`: Prefer high floor room
- `QUIET_ROOM`: Request quiet room away from elevator/lobby
- `TWIN_BEDS`: Request two separate beds instead of one
- `ACCESSIBILITY`: Wheelchair accessible room
- `DIETARY`: Dietary restrictions for breakfast
- `CELEBRATION`: Birthday/anniversary setup

6. GuestCount

Purpose: Encapsulate guest count dengan validation.

Attributes:

- `adults`: Integer (minimum 1)
- `children`: Integer (default 0)

Validation:

- Adults ≥ 1 (at least one adult required)
- Children ≥ 0
- Total does not exceed room capacity

Computed:

- `total()`: adults + children
- `requires_extra_bed()`: Check if extra bed needed

7. ReservationSource

Purpose: Track origin of reservation for analytics dan channel management.

Values:

- `WEBSITE`: Direct reservation via hotel website

- `MOBILE_APP`: Via hotel mobile app
- `PHONE`: Phone reservation
- `WALK_IN`: Walk-in reservation at front desk
- `OTA_RESERVATION_COM`: Via Reservation .com
- `OTA_AGODA`: Via Agoda
- `OTA_TRAVELOKA`: Via Traveloka
- `OTA_EXPEDIA`: Via Expedia
- `CORPORATE`: Corporate reservation (company contract)
- `TRAVEL_AGENT`: Via travel agency

Usage:

- Commission calculation (OTAs charge 15-25%)
- Channel performance analysis
- Marketing attribution

5.1.3 Domain Services

Domain Services contain business logic that:

- Doesn't naturally belong to any single aggregate
- Operates on multiple aggregates
- Represents a business process or calculation

1. AvailabilityService

Purpose: Complex availability queries dan analysis yang involve multiple Availability aggregates.

Responsibilities:

- Search available rooms across date ranges
- Calculate occupancy rates
- Predict demand for future periods
- Analyze reservation patterns

Key Operations:

search_availability(date_range, room_type_id, guest_count):

- Query availability for each date in range

- Check continuous availability (all dates available)
- Filter by room capacity if guest_count provided
- Return list of available room types dengan counts

calculate_occupancy_rate(date, room_type_id):

- Get total rooms for room type
- Get reserved rooms for date
- Calculate: $(\text{reserved} / \text{total}) \times 100\%$
- Used for revenue management decisions

predict_demand(date_range, room_type_id):

- Analyze historical reservation data for same period
- Consider trends (reservations increasing/decreasing)
- Factor in seasonality
- Return: DemandLevel (LOW, MEDIUM, HIGH)
- Used by Pricing Context untuk rate adjustments

is_high_demand_period(date_range):

- Check historical occupancy for period
- Check for special events (conferences, holidays)
- Return boolean untuk overbooking decisions

2. ReservationValidationService

Purpose: Validate business rules that require external knowledge or complex logic.

Responsibilities:

- Validate reservation against business constraints
- Check policy compliance
- Enforce reservation rules

Key Validations:

validate_reservation_rules(reservation, room_type, date_range):

- Check minimum stay requirement
- Validate advance reservation limit
- Check blackout dates
- Validate guest count capacity
- Return: ValidationResult dengan errors list

check_minimum_stay(date_range, room_type):

- Minimum stay varies by:
 - Room type (suite may require 2 nights minimum)
 - Day of week (weekend reservations require 2 nights)
 - Season (peak season requires 3 nights)
- Return boolean

check_blackout_dates(date_range):

- Blackout dates when reservation not allowed:
 - Hotel maintenance periods
 - Special events (private functions)
 - System upgrade windows
- Return boolean

check_advance_reservation_limit(check_in_date):

- Maximum 365 days in advance
- Some promotions may extend to 400+ days
- Prevents speculative reservations
- Return boolean

validate_guest_capacity(guest_count, room_type):

- Each room type has max capacity:
 - Standard: 2 adults + 1 child
 - Deluxe: 2 adults + 2 children
 - Suite: 4 adults + 2 children
 - Family: 6 adults + 4 children
- - Return boolean

3. OverbookingService

Purpose: Manage overbooking strategy untuk optimize occupancy while minimizing risk.

Responsibilities:

- Calculate safe overbooking limits
- Decide when to accept overbookings
- Manage overbooking situations (relocations)

Key Operations:

calculate_overbooking_limit(date, room_type):

- Analyze historical data:
 - No-show rate (typical 3-5% for hotels)
 - Cancellation rate by lead time
 - Day of week patterns (business travel Mon-Thu)
- Formula: ``limit = total_rooms × (no_show_rate + late_cancellation_rate)``
- Cap at 10% of total rooms (risk management)
- Return: Integer (number of rooms that can be overbooked)

should_accept_overbooking(date, room_type):

- Check current overbooking level
- Check demand level (only for high-demand)
- Check risk factors:
 - Weekend (lower no-show rate, risky)
 - Special events (everyone shows up, very risky)
 - Weather (affects no-shows)
- Return: Boolean decision

manage_overbooking_situation(date, room_type):

- Called when actual overbooking happens
- Find reservations that can be relocated:
 - OTA reservations (easier to relocate)
 - Non-loyalty members
 - Single-night stays
 - Later check-in times
 - Prioritize keeping:
 - Direct reservations
 - Loyalty members (Gold, Platinum)
 - Multi-night stays
 - Early check-ins
- - Return: List of relocatable reservations

calculate_relocation_cost(reservation):

- Cost to relocate guest to another hotel:
 - Room rate at comparable hotel
 - Transportation cost
 - Compensation (discount, points)
- Compare with penalty for refusing reservation
- Return: Money

6. TACTICAL DESIGN - RESERVATION CONTEXT

6.1 Aggregate Identification

Criteria untuk defining aggregates:

1. Transaction Boundary

- What must change together dalam single transaction?
- What invariants must be protected atomically?

- What consistency boundaries exist?

2. Concurrency Unit

- What gets locked together untuk prevent conflicts?
- What level of granularity untuk optimistic locking?
- Where do race conditions occur?

3. Lifecycle Boundary

- What is created dan destroyed together?
- What has independent lifecycle?
- What can exist without other entities?

4. Size Consideration

- Keep aggregates small untuk performance
- Minimize entities within aggregate
- Only include what's absolutely necessary

6.2 Identified Aggregates

Aggregate #1: Reservation (CORE)

Boundary:

- Reservation (Root Entity)
- DateRange (Value Object)
- GuestInfo (Value Object)
- PaymentInfo (Value Object)
- ReservationStatus (Value Object/Enum)

Design Rationale:

A. Why these entities together?

- Pertama, mereka protect critical business invariant yaitu reservation consistency. Semua perubahan pada dates, guest info, atau payment harus validated together untuk ensure validity.
- Kedua, they form transaction boundary. Create atau modify reservation requires semua components consistent dalam single transaction.
- Ketiga, concurrency control requirement. Ketika modify reservation, perlu lock entire reservation untuk prevent conflicts.
- Keempat, lifecycle dependency. DateRange, GuestInfo, dan PaymentInfo tidak make sense without Reservation context.

B. Why NOT include other entities?

- Availability counts managed separately karena they aggregate across many reservations. Including would make aggregate too large.
- Payment processing details managed oleh Payment Context. Reservation hanya track payment status, tidak payment internals.

- Room assignment (specific room number) managed oleh Operations Context. Reservation only cares about room type, not specific room.
- Guest profile details managed oleh Guest Context. Reservation hanya need minimal info untuk reservation.

Invariants Protected:

- Check-in date harus sebelum check-out date.
- Minimum stay duration 1 night.
- Cannot modify dates if already checked in.
- Total amount harus match calculated price.
- Status transitions must follow allowed paths (contoh: tidak bisa dari CheckedOut langsung ke Pending).
- Cannot cancel after check-in.
- Payment status must align dengan reservation status.

Aggregate #2: Availability

Boundary:

- Availability (Root only) - denormalized view

Design Rationale:

C. Why separate aggregate?

- Pertama, different consistency model. Availability tolerates eventual consistency, tidak perlu strong consistency seperti Reservation.
- Kedua, different performance characteristics. Availability queried frequently (every search), updated less frequently. Needs aggressive caching.
- Ketiga, aggregation across reservations. Availability is count dari many reservations. Doesn't belong ke any single reservation.
- Keempat, simpler concurrency model. Updates dapat be eventually consistent. Optimistic locking sufficient, tidak need pessimistic locks.

D. Why NOT inside Reservation?

Availability spans multiple reservations dan room types. Would require querying all reservations untuk calculate - performance nightmare. Denormalized view provides fast lookups. Updates propagated via events - eventual consistency acceptable. Real-time updates not critical karena seconds delay OK.

Invariants Protected:

Total rooms sama dengan available plus reserved plus blocked (accounting). Cannot reserve more than total plus overbooking limit. Blocked rooms cannot be reserved. Counts cannot be negative.

Aggregate #3: Waitlist

Boundary:

- Waitlist (Root only)

Design Rationale:**E. Why separate aggregate?**

- Pertama, independent lifecycle from Reservation. Waitlist entries dapat exist tanpa reservation. Conversion to reservation adalah separate transaction.
- Kedua, can be processed asynchronously. Waitlist has lower priority than core reservation flow. Different business rules dan policies.

Ketiga, may remain in waitlist indefinitely. Different stakeholders manage it.

F. Why NOT inside Reservation?

Waitlist entries exist without reservation. Conversion to reservation adalah separate transaction. May remain in waitlist indefinitely. Different stakeholders manage it.

Invariants Protected:

Waitlist entry must have valid dates. Priority must be valid enum value. Cannot convert expired waitlist. One guest can have multiple waitlist entries.

6.3 Domain Events Strategy

Event Publishing Pattern - Outbox Pattern**Why Outbox Pattern?**

- **Reliability:** Events guaranteed to be published. Tidak ada event yang lost even jika message broker down.
- **Consistency:** Events saved dalam same transaction sebagai aggregate. Database transaction ensures atomicity.
- **Idempotency:** Can retry safely. Duplicate events dapat di-handle oleh consumers.
- **Audit trail:** All events persisted. Complete history untuk debugging dan compliance.
- **At-least-once delivery:** Ensures no events lost. Better than at-most-once.

Event Flow:

- Pertama, aggregate change happens dalam domain layer.
- Kedua, domain event saved ke outbox table dalam same transaction dengan aggregate changes.
- Ketiga, background worker polls outbox table periodically.
- Keempat, publish event ke message broker seperti RabbitMQ atau Kafka.
- Kelima, mark as published dalam outbox setelah successful publish.
- Keenam, consumers process event dari message broker.

Event Consumers

Pricing Context consumes:

- ReservationCreated untuk track demand for forecasting
- ReservationCancelled untuk update cancellation statistics

Operations Context consumes:

- GuestCheckedIn untuk trigger operational workflows
- ReservationConfirmed untuk prepare for arrival

Guest Context consumes:

- GuestCheckedOut untuk update stay history dan award points

Event Versioning Strategy

Challenge: Events persisted dan consumed by multiple contexts. Bagaimana handle schema changes?

Solution: Event versioning dengan upcasting.

Setiap event memiliki version field. Old consumers dapat process old versions tanpa breaking. New consumers upcast old events ke new schema otomatis. Backward compatibility maintained untuk smooth migrations.

Example:

1. Version 1 menggunakan separate check_in_date dan check_out_date fields.
2. Version 2 menggunakan combined DateRange value object.
3. Upcaster converts V1 ke V2 ketika reading old events dari event store.

6.4 Repository Implementation Strategy

Separation of Concerns:

Domain layer defines repository interface sebagai abstraction. Tidak ada implementation details di domain. Infrastructure layer provides concrete implementation. Domain tidak depend on infrastructure details.

Benefits:

- **Testability:** Can mock repository dalam domain tests. Tidak perlu real database untuk unit tests.

- **Flexibility:** Can swap database implementation. Switch dari PostgreSQL ke MongoDB jika needed.
- **Clean architecture:** Domain independent dari infrastructure. Business logic tidak coupled dengan persistence.
- **Multiple implementations:** In-memory repository untuk tests. PostgreSQL repository untuk production. Different implementations untuk different environments.

Repository Responsibilities:

Persistence dan retrieval of aggregates. Mapping between domain objects dan database schema. Transaction management dan commit/rollback. Query optimization untuk performance. Connection pooling management.

Not Repository Responsibilities:

Business logic - belongs dalam domain entities dan services. Complex queries across aggregates - use application service. Direct aggregate-to-aggregate references - use IDs only. Validation logic - belongs dalam domain.

7. STRATEGI IMPLEMENTASI

7.1 Technology Stack

Backend Framework: FastAPI (Python)

Justification:

- **Async support:** Native async/await untuk high concurrency. Handles many simultaneous requests efficiently.
- **Type hints:** Strong typing aligns dengan DDD value objects. Static type checking catches errors early.
- **Performance:** Comparable ke Node.js performance. Faster than Django untuk API workloads.
- **API documentation:** Auto-generated OpenAPI/Swagger docs. Reduces documentation burden.
- **Dependency injection:** Built-in DI container. Clean architecture implementation.
- **Validation:** Pydantic models untuk request/response validation. Type-safe data validation.

Primary Database: PostgreSQL

Justification:

- **ACID transactions:** Strong consistency untuk aggregates. Critical untuk maintaining invariants.
- **JSON support:** Store event data dalam outbox table. Flexible schema untuk events.
- **Performance:** Excellent untuk transactional workloads. Battle-tested dalam production.
- **Mature ecosystem:** Banyak tools dan libraries available. Large community support.

- **Extensions:** PostGIS untuk future location features. Full-text search built-in.
- **Caching Layer:** Redis

Use Cases:

Cache availability data untuk hot path optimization. Session management untuk user sessions. Rate limiting untuk API protection. Distributed locks untuk concurrency control. Temporary holds during reservation process dengan 5 minute TTL.

Message Broker: RabbitMQ

Justification:

Simpler than Kafka untuk getting started. Lower learning curve. **Good for request/response** patterns dengan RPC support. **Transactional messaging** support untuk reliability. **Lower operational complexity** compared to Kafka. **Mature and stable** dengan proven track record.

Alternative: Kafka for high-throughput event streaming di future jika needed.

API Gateway: Kong

Features:

Rate limiting per client untuk prevent abuse. Authentication/Authorization centralized. Request/Response transformation untuk versioning. API versioning support untuk backward compatibility. Monitoring dan logging centralized. Circuit breaker patterns untuk fault tolerance.

7.2 Data Management Strategy

Database Per Context:

Reservation Database berisi:

- Tabel reservations untuk main reservation data
- Tabel availability untuk denormalized counts
- Tabel waitlist untuk pending requests
- Tabel outbox_events untuk event publishing

Inventory Database berisi:

- Tabel rooms untuk physical room data
- Tabel room_types untuk room categories
- Tabel maintenance_schedules untuk downtime planning

Pricing Database berisi:

- Tabel rates untuk base pricing
- Tabel pricing_rules untuk dynamic pricing logic
- Tabel historical_rates untuk analytics

Guest Database berisi:

- Tabel guest_profiles untuk customer data
- Tabel preferences untuk personalization
- Tabel loyalty_accounts untuk rewards program

Payment Database berisi:

- Tabel transactions untuk payment records
- Tabel invoices untuk billing
- Tabel refunds untuk cancellation handling

Benefits:

- **Autonomy:** Each context evolves independently tanpa coordination overhead.
- **Scalability:** Scale databases independently based on load patterns.
- **Fault isolation:** Failure dalam one DB doesn't affect others. System remains partially available.
- **Technology diversity:** Different DBs untuk different needs jika appropriate.

Challenges:

- **No foreign keys** across contexts. Data integrity enforced dalam application layer.
- **Eventual consistency** untuk cross-context data. Requires careful design.
- **Data duplication** untuk denormalization. Trade consistency untuk performance.

CQRS Pattern Implementation:

Write Model (Command Side):

Domain model dengan business logic. Strong consistency requirements. Optimized untuk writes dan validations. Command handlers process user actions.

Read Model (Query Side):

Denormalized views untuk fast queries. Eventual consistency acceptable. Optimized untuk read performance. Materialized views updated via events.

Benefits:

- **Performance:** Separate optimization strategies untuk reads vs writes.
- **Scalability:** Scale reads dan writes independently based on demand.
- **Simplicity:** Query model doesn't need domain complexity. Just flat data structures.
- **Synchronization:** Domain events update read models asynchronously. Event handlers maintain denormalized data. Background jobs untuk reconciliation jika needed.

7.3 Scalability Architecture

Horizontal Scaling Strategy:

- **Load Balancer Configuration:**

Internet traffic masuk ke Load Balancer seperti HAProxy atau Nginx. Load balancer distribute requests ke multiple application instances. Misalnya App 1, App 2, dan App 3 yang merupakan FastAPI instances. Application instances connect ke PostgreSQL database setup dengan Primary Replica configuration.

- **PostgreSQL Primary-Replica Setup:**

Primary database handle semua write operations. Write operations require strong consistency. Replica databases handle read operations. Misalnya Replica 1, Replica 2, dan Replica 3 untuk distribute read load. Read operations can scale horizontally.

8. Kesimpulan

Berdasarkan analisis domain yang telah dilakukan, telah diidentifikasi dan didefinisikan 6 Bounded Context utama untuk Sistem Reservasi Hotel. Konteks-konteks ini diklasifikasikan berdasarkan nilai strategisnya, yaitu Core/Inti (Reservation Context, Pricing Context), Supporting/Pendukung (Inventory Context, Guest Context, Operations Context), dan Strategic/Strategis (Payment Context).

Untuk memvisualisasikan interaksi dan dependensi antar konteks, sebuah Context Map telah disusun. Peta ini merinci pola relasi yang digunakan, seperti Anti-Corruption Layer (ACL) untuk melindungi domain dari kompleksitas Pricing dan sistem Payment eksternal, pola Open Host Service (OHS) yang disediakan oleh Inventory, relasi Customer/Supplier antara Reservation dan Guest, serta Shared Kernel antara Inventory dan Operations untuk sinkronisasi status kamar secara *real-time*.

Setelah melalui analisis komparatif yang mengevaluasi kriteria seperti nilai bisnis, kompleksitas, dan dampak pengguna, **Reservation Context [CORE]** dipilih sebagai Konteks Inti (Core Domain). Pemilihan ini didasarkan pada justifikasi bahwa Reservation Context merupakan jantung bisnis, penggerak utama konversi pemesanan, dan berfungsi sebagai hub integrasi teknis yang paling kompleks dalam keseluruhan sistem.

Sebagai tindak lanjut dari desain strategis ini, dokumen Milestone 3 (Desain Taktis) dan Milestone 4 (Implementasi Kode) akan memusatkan fokus pengembangan **eksklusif pada Reservation Context**. Bounded Context lainnya (Pricing, Inventory, Guest, Operations, Payment) hanya akan didefinisikan sebatas kontrak antarmuka (*interface contracts*) dan simulasi pertukaran *domain events* untuk mendemonstrasikan integrasi, tanpa implementasi logika internal yang mendalam. Pendekatan ini diambil untuk memastikan kualitas dan kedalaman implementasi pada konteks inti utama sebelum meluaskan cakupan sistem.

9. Daftar Pustaka

Domain-Driven Design (DDD). (2022). Slide kuliah II3160 – Teknologi Sistem Terintegrasi, STEI ITB.

Service-Oriented Architecture (SOA). (2022). Slide kuliah II3160 – Teknologi Sistem Terintegrasi, STEI ITB.

Microservice Architecture. (2022). Slide kuliah II3160 – Teknologi Sistem Terintegrasi, STEI ITB.

Architectural Styles, Architectural Patterns, and Design Patterns. (2022). Slide kuliah II3160 – Teknologi Sistem Terintegrasi, STEI ITB.

Application Programming Interface (API). (2022). Slide kuliah II3160 – Teknologi Sistem Terintegrasi, STEI ITB.

Web System Design (Scaling, Load Balancer, Cache, MQ). (2022). Slide kuliah II3160 – Teknologi Sistem Terintegrasi, STEI ITB.

Systems Integration (Arsitektur & Strategi Integrasi). (2022). Slide kuliah II3160 – Teknologi Sistem Terintegrasi, STEI ITB.

Evans, E. (2003). *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley.

Vernon, V. (2013). *Implementing Domain-Driven Design*. Addison-Wesley.

Fielding, R. T. (2000). *Architectural Styles and the Design of Network-based Software Architectures* (Doctoral dissertation, UC Irvine).

Fowler, M. (2002). *Patterns of Enterprise Application Architecture*. Addison-Wesley.

Hohpe, G., & Woolf, B. (2003). *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley.

Richardson, C. (2018). *Microservices Patterns: With Examples in Java*. Manning.

Newman, S. (2021). *Building Microservices* (2nd ed.). O'Reilly.

Kleppmann, M. (2017). *Designing Data-Intensive Applications*. O'Reilly.