Full Stack Development Assignment

Name: Ashwin Kumar S

Register No: 3122225002014

FinTech Vertical:

FinApp — a payments + wallets + merchant settlement platform used by consumers and merchants. Features: user onboarding/KYC, wallet top-up, payments (P2P & merchant), refunds, settlement to merchant bank accounts, fraud detection, notifications, reporting.

Identified bounded contexts (DDD):

By analysing the domain and split into meaningful bounded contexts (each with ubiquitous language and independent model):

1. **Customer Management** — users, KYC status, authentication, profile.
2. **Wallet & Accounts** — wallet balance, ledger entries, top-ups, withdrawals.
3. **Payments** — payment initiation, routing (card/netbanking), payment status.
4. **Settlement** — batching merchant payouts to bank accounts, settlement reconciliation.
5. **Risk & Fraud** — scoring, rules, blocks, alerts.
6. **Notifications** — email/SMS/push templates and dispatch.
7. **Reporting & Audit** — transaction history, compliance reporting, ledgers.
8. **Billing & Fees** — merchant fee calculation, invoicing.
9. **Dispute & Chargeback** — disputes lifecycle, evidence collection.
10. **Compliance / KYC Ops** — manual review workflows.

Each bounded context uses different models: e.g., “Balance” in Wallet context vs “LedgerEntry” in Reporting context.

DDD artifacts:

- **Aggregate** (Wallet context): Wallet aggregate root; enforces invariants (balance >= 0 unless credit allowed).
- **Entities**: User, Merchant, Transaction.

- **Value Objects**: Money(amount,currency), AccountNumber.
- **Repositories**: WalletRepository, TransactionRepository.
- **Domain Events**: PaymentInitiated, PaymentSucceeded, PaymentFailed, FundsReserved, FundsReleased, SettlementBatchCreated.

Event design (EDA) — event catalog :

- KYCCompleted { userId, kycLevel }
- WalletToppedUp { walletId, amount, source }
- PaymentInitiated { paymentId, amount, fromWallet, toMerchant }
- PaymentSucceeded { paymentId, settledAt }
- FraudAlertRaised { paymentId, riskScore }
- SettlementCreated { batchId, merchantId, amount }

Events are **facts** (immutable) and used to drive reactions (e.g., Notifications service subscribes to PaymentSucceeded to send receipts).

Microservice decomposition (MSA):

Map bounded contexts to microservices (one service per context typically):

- customer-service (Customer Management) — stores user profiles, KYC status.
- wallet-service (Wallet & Accounts) — owns wallet balances and ledger entries.
- payment-service (Payments) — orchestrates payment flows, interacts with PSPs.
- settlement-service (Settlement) — batches merchant payouts.
- fraud-service (Risk & Fraud) — evaluates transactions, publishes FraudAlert.
- notification-service (Notifications) — subscribes to events to send messages.
- reporting-service (Reporting & Audit) — materialized views for analytics.
- billing-service (Billing & Fees) — computes fees and invoices.
- dispute-service (Dispute & Chargebacks).
- kyc-ops-service (Compliance manual review).

Each microservice:

- **Owns its data** (DB per service).
- **Communicates via**: (a) synchronous APIs for request/response, (b) asynchronous events for eventual consistency and decoupling.

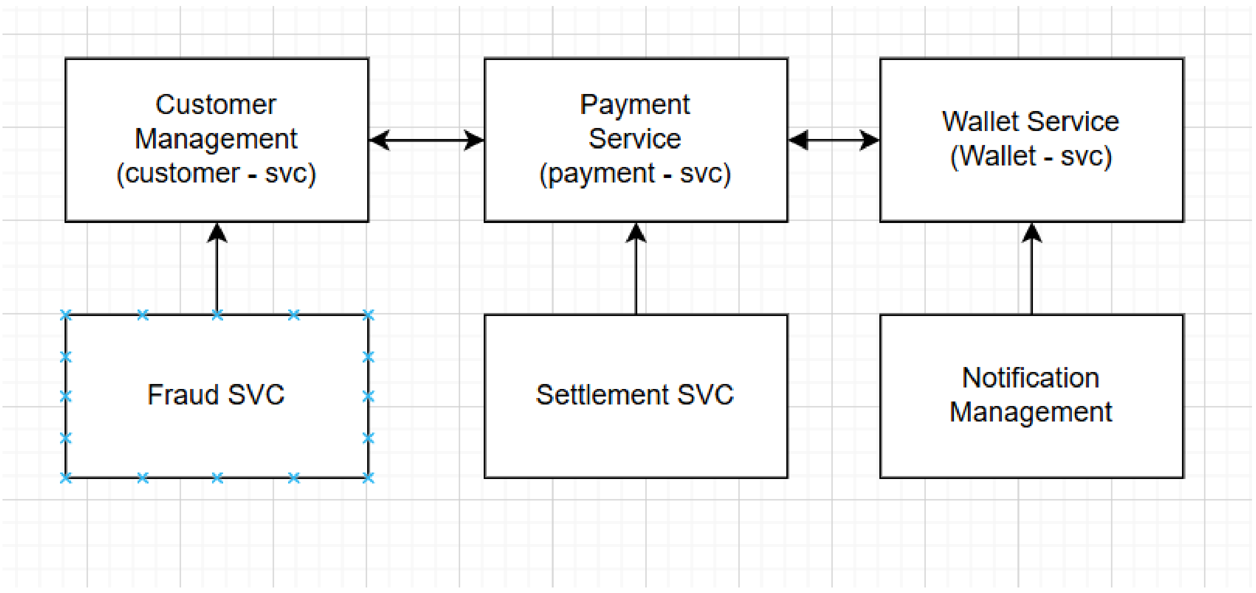
Interaction patterns:

A. Simple payment flow (asynchronous preferred)

1. Client calls payment-service to create payment → payment-service validates and emits PaymentInitiated.
2. wallet-service subscribes to PaymentInitiated, reserves funds (creates ledger entry with status reserved). On success emits FundsReserved.
3. payment-service listens for FundsReserved then calls PSP; on PSP success emits PaymentSucceeded.
4. wallet-service on PaymentSucceeded finalizes ledger (debits), emits WalletDebited.
5. notification-service on PaymentSucceeded sends receipt; reporting-service updates analytics.

B. Settlement flow (batch)

- payment-service emits PaymentSettled events. settlement-service aggregates by merchant into a SettlementBatchCreated event; then triggers bank transfers.


Customer Management (customer - svc)
Payment Service (payment - svc)
Wallet Service (Wallet - svc)
Fraud SVC
Settlement SVC
Notification Management

Interaction Patterns and Flow

Data consistency and transactions:

- Use **sagas (process managers)** for cross-service workflows: e.g., payment saga coordinates PaymentInitiated → reserve → charge → finalize; compensating actions for failure (release funds).
- Prefer **eventual consistency** for performance and availability.
- For strong consistency within a service, use ACID DB transactions (local).

Non-functional Requirements (FinTech specifics):

- **Security & Compliance:** encryption-at-rest/in-transit, secrets management, audit trails, role-based access, KYC/GDPR/PFMI/PCI-DSS considerations. Services generating events must redact PII in events or use tokenized references.
- **Observability:** distributed tracing (e.g., trace id across services), centralized logging, metrics (latency, error rates). Critical for fraud detection and incident response.
- **Resilience:** retry policies, idempotency (events and commands must be idempotent), circuit breakers for PSP calls.
- **Scaling:** payment and wallet services scale independently (high throughput during sales/holiday spikes).
- **Latency:** synchronous paths kept minimal; move expensive processing (reporting/ML scoring) to async consumers.

EdTech vertical:

UseCase:

An online learning platform: course catalog, enrollment, content delivery, quizzes, grading, notifications, analytics.

Bounded contexts:

- **User & Auth** (students, instructors).
- **Course Catalog** (courses, modules).
- **Learning Progress** (enrollments, completion, progress checkpoints).
- **Assessment** (quizzes, grading).
- **Recommendation / Analytics** (personalized suggestions, dashboards).
- **Notification**.

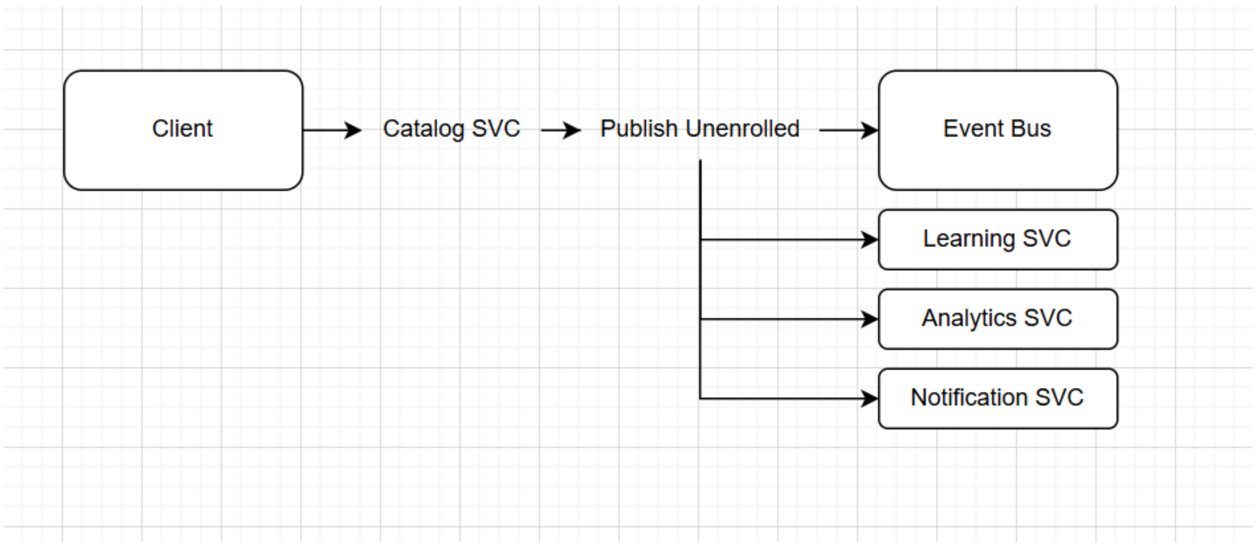
DDD → Microservices mapping:

- user-service

- catalog-service
- learning-service
- assessment-service
- analytics-service
- notification-service.

Event examples (EDA):

- UserEnrolled
- ModuleCompleted
- QuizSubmitted
- GradePublished
- RecommendationUpdated.



Impacts:

- **MSA + EDA** enables near-real-time progress tracking, decoupled grading and recommendation pipelines, horizontal scaling of content delivery; tradeoffs include operational overhead and testing complexity.

Impact analysis:

Positive impacts:

- **Alignment to business**: DDD produces clear service boundaries matching business teams; reduces cognitive load.
- **Loose coupling & scalability**: MSA + EDA allow independent scaling and deployment; high-throughput event processing handles spikes.
- **Resilience & fault isolation**: failure in one service (e.g., reporting) doesn't bring down payments if designed correctly.

- **Faster team delivery**: teams own bounded contexts (dev/ops autonomy).
- **Real-time capabilities**: EDA enables instant notifications, dashboards, and streaming analytics.
- **Easier evolution**: new features are added as services/events without modifying monolithic codebase.

Drawbacks / tradeoffs:

- **Operational complexity**: distributed systems require sophisticated DevOps, monitoring, and CI/CD.
- **Data consistency**: eventual consistency can be surprising; requires sagas and compensating transactions.
- **Testing complexity**: integration and contract testing required (consumer-driven contracts).
- **Latency & debugging**: tracing flows across async events more complex.
- **Event schema management**: event versions and compatibility must be managed.
- **Cost**: multiple runtimes, infra, and data stores increase cost.

FinTech specific tradeoffs:

- **Regulatory constraints** may require stronger consistency and audit trails — sometimes favor synchronous flows for money movement.
- **Security overhead** is higher (PCI, KYC), so event payloads must avoid exposed secret

Practical patterns & recommendations:

1. **Bounded-contexts first** — use DDD workshop with domain experts to define contexts and ubiquitous language before splitting services.
2. **Single responsibility per service** — keep services small but meaningful; avoid over-fragmentation.
3. **Event schema governance** — use schema registry (Avro/Protobuf) and versioning practices.
4. **Sagas for multi-step workflows** — implement orchestrator (central saga) or choreography (events) depending on complexity.
5. **Idempotency & deduplication** — idempotent handlers for events and commands.
6. **Use CQRS where needed** — separate read models (materialized views) for high-performance queries.
7. **Security & Compliance by design** — encrypt sensitive data, mask PII in events, maintain audit trails.

8. **Observability** — distributed tracing, correlation IDs, centralized logs and metrics.
9. **Testing** — unit tests + contract tests + end-to-end with test harness for event buses.
10. **Operational automation** — infra as code, automated rollback, blue/green or canary deployments.

Conclusion:

Using **DDD** to discover bounded contexts gives a solid semantic map for service boundaries. Implementing those contexts as **microservices (MSA)** provides deployment independence and team autonomy; coupling MSA with **EDA** enables reactive, loosely coupled systems suited for real-time needs such as payments, notifications, or learning progress. The trio together is powerful for complex domains (FinTech, EdTech) but requires disciplined design — especially event schema governance, compensating transactions (sagas), strong observability, and security/compliance practices.