**📘 Payment Oversight Module – Technical Documentation (Squad E3.4)**

**1. System Architecture**

**1.1 Overview**

The **Payment Oversight Module** ensures secure handling of financial transactions by:

* Detecting and managing **flagged transactions** (potentially suspicious).
* Enabling **payment dispute resolution** through structured workflows.
* Maintaining a full **audit trail** for transparency and compliance.

**Core Principles**:

* Security-first design.
* Regulatory compliance (AML/KYC, GDPR).
* Horizontal scalability.
* End-to-end logging and monitoring.

**1.2 High-Level Architecture**

* **Frontend**: Admin dashboard (Angular/React).
* **Backend Services**: Flask/Django REST-based APIs.
* **Database Layer**: PostgreSQL + Redis for caching flagged transactions.
* **Notification Service**: Email/SMS/Push integration.
* **Audit Logging Service**: Immutable append-only logs (stored in PostgreSQL or ELK stack).
* **Monitoring**: Prometheus + Grafana dashboards.

flowchart LR

A[Admin User] --> B[Admin Dashboard UI]

B --> C[Payments Oversight API]

C --> D[(Database: flagged\_transactions, payment\_disputes, audit\_logs)]

C --> E[Notification Service]

C --> F[Audit Logging Service]

**2. Data Model Detailed Specifications**

**2.1 flagged\_transactions**

| **Field** | **Type** | **Constraints** | **Description** |
| --- | --- | --- | --- |
| id | UUID | PK | Unique identifier |
| transaction\_id | UUID | FK → transactions | Associated transaction |
| flagged\_reason | TEXT | NOT NULL | Reason flagged (AML, fraud, anomaly) |
| flagged\_at | TIMESTAMP | DEFAULT now() | Time flagged |
| status | ENUM | {pending, reviewed, cleared} | Flag status |

**2.2 payment\_disputes**

| **Field** | **Type** | **Constraints** | **Description** |
| --- | --- | --- | --- |
| id | UUID | PK | Unique identifier |
| transaction\_id | UUID | FK → transactions | Disputed transaction |
| user\_id | UUID | FK → users | Disputing user |
| dispute\_reason | TEXT | NOT NULL | Dispute details |
| status | ENUM | {open, under\_review, resolved, escalated} | Lifecycle state of dispute |
| created\_at | TIMESTAMP | DEFAULT now() | Created timestamp |
| resolved\_at | TIMESTAMP | NULLABLE | Resolution timestamp |

**2.3 audit\_logs**

| **Field** | **Type** | **Constraints** | **Description** |
| --- | --- | --- | --- |
| id | UUID | PK | Unique identifier |
| admin\_id | UUID | FK → admins | Acting admin |
| action\_type | ENUM | {flagged\_view, dispute\_resolved, escalation, notification\_sent} | Admin action |
| target\_id | UUID | FK → {flagged\_transactions, payment\_disputes} | Affected entity |
| timestamp | TIMESTAMP | DEFAULT now() | Action timestamp |
| metadata | JSONB |  | Extra contextual details |

**3. API Contract and Endpoint Definitions**

**3.1 GET /api/v1/admin/payments/flagged**

**Description**: Retrieve all flagged transactions pending admin review.

**Response Example**:

[

{

"id": "uuid-123",

"transaction\_id": "txn-789",

"flagged\_reason": "Unusual amount",

"status": "pending",

"flagged\_at": "2025-09-10T14:22:11Z"

}

]

**Error Codes**:

* 401 Unauthorized – Invalid admin token.
* 404 Not Found – No flagged transactions available.
* 500 Internal Server Error.

**3.2 POST /api/v1/admin/payments/disputes/{id}/resolve**

**Description**: Resolve a payment dispute.

**Request Body**:

{

"resolution": "Transaction refunded",

"resolved\_by": "admin-456"

}

**Response Example**:

{

"id": "dispute-123",

"status": "resolved",

"resolved\_at": "2025-09-16T10:00:00Z"

}

**Error Codes**:

* 400 Bad Request – Invalid input data.
* 404 Not Found – Dispute ID not found.
* 409 Conflict – Dispute already resolved.
* 500 Internal Server Error.

**4. Dispute Resolution Workflow**

sequenceDiagram

participant Admin

participant API

participant DB

participant User

participant AuditLog

Admin->>API: View flagged transactions

API->>DB: Fetch flagged records

API-->>Admin: Return flagged list

Admin->>API: Resolve dispute (POST)

API->>DB: Update dispute status

API->>User: Notify resolution outcome

API->>AuditLog: Record admin action

**5. Security and Compliance Considerations**

* **Encryption**: AES-256 for data at rest, TLS 1.3 for data in transit.
* **Access Control**: RBAC with JWT-based admin authentication.
* **Compliance**: GDPR, AML/KYC adherence.
* **Audit Trail**: Immutable logging (append-only).
* **PII Masking**: Sensitive user data masked in logs.

**6. Error Handling Strategies**

* Use structured JSON error responses:

{

"error\_code": "DISPUTE\_NOT\_FOUND",

"message": "The requested dispute does not exist."

}

* Graceful degradation for partial failures.
* Retry + backoff for failed notifications.
* Alerting for repeated error patterns (via Prometheus).

**7. Performance Optimization Techniques**

* Indexes on transaction\_id and status fields.
* Database connection pooling.
* Redis caching for frequently accessed flagged transactions.
* Asynchronous job queues (Celery/RQ) for notifications.

**8. Comprehensive Testing Strategy**

**8.1 Unit Tests**

* Dispute resolution logic (status transitions).
* Validation of flagged transaction schema.

**8.2 Integration Tests**

* End-to-end flagged transaction lifecycle.
* Escalation workflow with notifications.

**8.3 Negative Tests**

* Invalid dispute/transaction IDs.
* Unauthorized admin attempts.

**8.4 Performance Tests**

* Stress test with 10k flagged transactions.
* Load test dispute resolution concurrency.

**9. Deployment and Scaling Recommendations**

* **Containerization**: Docker + Kubernetes.
* **CI/CD**: GitHub Actions + Helm for deployments.
* **Monitoring**: Alerts for abnormal dispute/flag rates.
* **Scaling**: Horizontal scaling of API services.
* **Future Enhancements**:
  + ML-based fraud detection for auto-flagging.
  + Chatbot-based dispute query system.
  + Blockchain-based immutable audit logs.

**🔧 Troubleshooting & Best Practices**

* Always check **audit logs** for dispute inconsistencies.
* Ensure **flagged transaction reasons** are human-readable.
* Use **escalation workflows** for disputes older than SLA.
* Enable **rate limiting** to prevent brute-force attempts.

✅ **This documentation covers everything needed** for architecture, APIs, workflows, testing, compliance, and deployment.

| **Week** | **Deliverable** |
| --- | --- |
| 1 | Database schema design & migrations |
| 2–3 | Flagged transaction API implementation |
| 4 | Dispute resolution API implementation |
| 5 | Escalation workflow integration |
| 6 | Comprehensive test coverage |
| 7 | Performance tuning & stress tests |
| 8 | Final documentation & demo |

Admin → API → DB → User Notification → Audit Logs

Perfect — let’s translate the **Squad E3.4 Payments Oversight MVP** into a **FastAPI implementation** 🚀

This will include:

* **SQLAlchemy models** (flagged\_transactions, payment\_disputes, audit\_logs)
* **Pydantic schemas**
* **API endpoints** (GET flagged txns, POST resolve dispute)
* **Background tasks** for async notifications
* **Alembic migrations** (schema design)
* **Pytest tests**
* **Docker + docker-compose** for dev setup
* **OpenAPI auto-docs** at /docs
* **Architecture diagrams + sequence flow**

**📘 FastAPI Payments Oversight (Squad E3.4)**

**📂 Project structure**

fastapi\_payments\_oversight/

├── app/

│ ├── \_\_init\_\_.py

│ ├── main.py

│ ├── config.py

│ ├── database.py

│ ├── models.py

│ ├── schemas.py

│ ├── crud.py

│ ├── api.py

│ ├── tasks.py

│ ├── deps.py

│ └── tests/

│ └── test\_disputes.py

├── alembic/

│ └── versions/

├── alembic.ini

├── requirements.txt

├── Dockerfile

├── docker-compose.yml

└── README.md

**⚙️ requirements.txt**

fastapi

uvicorn[standard]

SQLAlchemy

alembic

psycopg2-binary

pydantic

python-dotenv

httpx

pytest

pytest-asyncio

**🛠 app/config.py**

import os

from dotenv import load\_dotenv

load\_dotenv()

DATABASE\_URL = os.getenv("DATABASE\_URL", "postgresql+psycopg2://dev:dev@db:5432/devdb")

**🗄 app/database.py**

from sqlalchemy import create\_engine

from sqlalchemy.orm import sessionmaker, declarative\_base

from .config import DATABASE\_URL

engine = create\_engine(DATABASE\_URL, future=True)

SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)

Base = declarative\_base()

**🧩 app/models.py**

import uuid

from sqlalchemy import Column, String, DateTime, Text, ForeignKey, JSON

from sqlalchemy.dialects.postgresql import UUID

from sqlalchemy.sql import func

from .database import Base

class FlaggedTransaction(Base):

\_\_tablename\_\_ = "flagged\_transactions"

id = Column(UUID(as\_uuid=True), primary\_key=True, default=uuid.uuid4)

txn\_id = Column(UUID(as\_uuid=True), index=True, nullable=False)

reason = Column(Text, nullable=False)

status = Column(String(20), default="pending", index=True)

created\_at = Column(DateTime(timezone=True), server\_default=func.now())

metadata = Column(JSON, nullable=True)

class PaymentDispute(Base):

\_\_tablename\_\_ = "payment\_disputes"

id = Column(UUID(as\_uuid=True), primary\_key=True, default=uuid.uuid4)

txn\_id = Column(UUID(as\_uuid=True), index=True, nullable=False)

user\_id = Column(UUID(as\_uuid=True), index=True, nullable=False)

status = Column(String(20), default="open", index=True)

created\_at = Column(DateTime(timezone=True), server\_default=func.now())

resolved\_at = Column(DateTime(timezone=True), nullable=True)

resolution\_notes = Column(Text, nullable=True)

class AuditLog(Base):

\_\_tablename\_\_ = "audit\_logs"

id = Column(UUID(as\_uuid=True), primary\_key=True, default=uuid.uuid4)

admin\_id = Column(UUID(as\_uuid=True), nullable=True, index=True)

action = Column(String(32), index=True)

details = Column(JSON, nullable=True)

created\_at = Column(DateTime(timezone=True), server\_default=func.now())

**📑 app/schemas.py**

import uuid

from pydantic import BaseModel

from datetime import datetime

from typing import Optional, Dict

class FlaggedTransactionBase(BaseModel):

txn\_id: uuid.UUID

reason: str

status: str

metadata: Optional[Dict] = None

class FlaggedTransactionOut(FlaggedTransactionBase):

id: uuid.UUID

created\_at: datetime

class Config: orm\_mode = True

class PaymentDisputeOut(BaseModel):

id: uuid.UUID

txn\_id: uuid.UUID

user\_id: uuid.UUID

status: str

created\_at: datetime

resolved\_at: Optional[datetime]

resolution\_notes: Optional[str]

class Config: orm\_mode = True

class DisputeResolveIn(BaseModel):

resolution: str

resolved\_by: uuid.UUID

resolution\_notes: Optional[str] = None

**⚡ app/crud.py**

from sqlalchemy.orm import Session

from sqlalchemy import select

from . import models, schemas

from datetime import datetime

def get\_flagged\_transactions(db: Session):

return db.execute(select(models.FlaggedTransaction).order\_by(models.FlaggedTransaction.created\_at.desc())).scalars().all()

def get\_dispute(db: Session, dispute\_id):

return db.get(models.PaymentDispute, dispute\_id)

def resolve\_dispute(db: Session, dispute: models.PaymentDispute, data: schemas.DisputeResolveIn):

if dispute.status == "resolved":

return None

dispute.status = "resolved"

dispute.resolved\_at = datetime.utcnow()

dispute.resolution\_notes = data.resolution\_notes or data.resolution

db.add(dispute)

db.add(models.AuditLog(admin\_id=data.resolved\_by, action="dispute\_resolved", details={"dispute\_id": str(dispute.id), "resolution": data.resolution}))

db.commit()

db.refresh(dispute)

return dispute

**📡 app/api.py**

from fastapi import APIRouter, Depends, BackgroundTasks, HTTPException

from sqlalchemy.orm import Session

from . import schemas, crud, models

from .database import SessionLocal

from .tasks import notify\_user

router = APIRouter(prefix="/api/v1/admin/payments", tags=["payments"])

def get\_db():

db = SessionLocal()

try:

yield db

finally:

db.close()

@router.get("/flagged", response\_model=list[schemas.FlaggedTransactionOut])

def list\_flagged\_transactions(db: Session = Depends(get\_db)):

return crud.get\_flagged\_transactions(db)

@router.post("/disputes/{dispute\_id}/resolve", response\_model=schemas.PaymentDisputeOut)

def resolve\_dispute(dispute\_id: str, data: schemas.DisputeResolveIn, background: BackgroundTasks, db: Session = Depends(get\_db)):

dispute = crud.get\_dispute(db, dispute\_id)

if not dispute:

raise HTTPException(status\_code=404, detail="Dispute not found")

updated = crud.resolve\_dispute(db, dispute, data)

if not updated:

raise HTTPException(status\_code=409, detail="Dispute already resolved")

background.add\_task(notify\_user, str(updated.user\_id), str(updated.id), data.resolution)

return updated

**🔔 app/tasks.py**

import logging

logger = logging.getLogger("tasks")

def notify\_user(user\_id: str, dispute\_id: str, resolution: str):

# Here you could integrate email, SMS, or message queue

logger.info(f"Notify user {user\_id} about dispute {dispute\_id}: {resolution}")

**🚀 app/main.py**

from fastapi import FastAPI

from . import models, database, api

models.Base.metadata.create\_all(bind=database.engine)

app = FastAPI(title="Payments Oversight API", version="1.0.0")

app.include\_router(api.router)

Run with:

uvicorn app.main:app --reload

Docs at:

* Swagger UI → <http://127.0.0.1:8000/docs>
* ReDoc → <http://127.0.0.1:8000/redoc>

**🧪 Tests — app/tests/test\_disputes.py**

import pytest

from fastapi.testclient import TestClient

from app.main import app

from app.database import SessionLocal

from app import models

client = TestClient(app)

@pytest.fixture

def db():

db = SessionLocal()

yield db

db.close()

def test\_resolve\_dispute(db):

dispute = models.PaymentDispute(txn\_id="11111111-1111-1111-1111-111111111111", user\_id="22222222-2222-2222-2222-222222222222")

db.add(dispute)

db.commit()

db.refresh(dispute)

resp = client.post(f"/api/v1/admin/payments/disputes/{dispute.id}/resolve", json={

"resolution": "Refund issued",

"resolved\_by": "33333333-3333-3333-3333-333333333333"

})

assert resp.status\_code == 200

data = resp.json()

assert data["status"] == "resolved"

Run:

pytest -q

**🐳 Docker + docker-compose**

**Dockerfile**

FROM python:3.10-slim

WORKDIR /app

COPY requirements.txt .

RUN pip install --no-cache-dir -r requirements.txt

COPY . .

CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "8000"]

**docker-compose.yml**

version: "3.9"

services:

web:

build: .

ports:

- "8000:8000"

environment:

- DATABASE\_URL=postgresql+psycopg2://dev:dev@db:5432/devdb

depends\_on:

- db

db:

image: postgres:14

environment:

POSTGRES\_USER: dev

POSTGRES\_PASSWORD: dev

POSTGRES\_DB: devdb

ports:

- "5432:5432"

**🖼 Sequence Diagram (Mermaid)**

sequenceDiagram

Admin->>API: GET /api/v1/admin/payments/flagged

API->>DB: query flagged\_transactions

DB-->>API: results

API-->>Admin: list of flagged txns

Admin->>API: POST /api/v1/admin/payments/disputes/{id}/resolve

API->>DB: select dispute

API->>DB: update dispute -> resolved

API->>DB: insert audit\_log

API->>BackgroundTask: notify\_user()

BackgroundTask->>User: notification sent

API-->>Admin: 200 OK

✅ That’s the **FastAPI version** of the **Payments Oversight MVP** — lightweight, async-ready, with Alembic for migrations and background tasks for notifications.

**🔥 Payments Oversight Module (Explained Simply + Documented Deeply)**

**1. What It Does (Plain English)**

👉 This system is like a **financial watchdog**:

* It **spots shady transactions** (flagged).
* It **handles fights** between users and the platform (disputes).
* It **keeps receipts of every admin move** (audit logs).

In short → *“Catch it. Fix it. Record it.”*

**2. System Brains (Architecture)**

Think of it as layers:

1. **Admin Dashboard (React/Angular)** → Where admins click buttons.
2. **FastAPI Backend** → The logic brain.
3. **Database (Postgres + Redis)** → Where all money cases & logs are stored.
4. **Notification Service** → Tells users what’s happening.
5. **Audit Logging** → Keeps proof that nobody cheated.
6. **Monitoring (Prometheus/Grafana)** → Watchdog for the watchdog.

🖼 **Diagram**:  
Admin → Dashboard → API → Database → Notify User → Audit Log.

**3. Key Tables (Database Models)**

* **flagged\_transactions** → Suspicious payments.
* **payment\_disputes** → User complaints.
* **audit\_logs** → Who did what, when.

**4. API Endpoints**

* GET /api/v1/admin/payments/flagged → Show me all “sus” transactions.
* POST /api/v1/admin/payments/disputes/{id}/resolve → Mark a fight as solved.

**5. Flow of Events**

1. Admin sees flagged transaction.
2. Admin clicks “resolve dispute”.
3. System updates DB.
4. User gets a notification.
5. Audit log entry created.

That’s it — **a 5-step dance**.

**6. Security**

* Data locked with **AES-256** + **TLS 1.3**.
* Only admins with JWT tokens can act.
* All moves logged forever.
* No sensitive data leaks → masked in logs.

**7. Testing Strategy**

* **Unit** → Check dispute resolution works.
* **Integration** → End-to-end flagged → resolved.
* **Negative** → Fake IDs, double resolving, invalid tokens.
* **Performance** → Handle 10k disputes fast.

**8. Timeline (8 Weeks)**

* **Week 1**: Database ready.
* **Week 2–3**: Flagged API built.
* **Week 4**: Disputes solved API.
* **Week 5**: Escalation flow.
* **Week 6**: Tests, tests, tests.
* **Week 7**: Speed-up + stress test.
* **Week 8**: Docs + Demo 🎉

**9. Why Client Should Care**

* Transparent: every admin move is traceable.
* Secure: meets finance-grade compliance.
* Scalable: can grow as transaction volume grows.
* Future-proof: ready for ML fraud detection & blockchain audit logs.