Each task require the appropriate level of automated testing

Task 1: Real-Time Multi-Tenant Event Feed

Problem

Build a real-time event broadcasting system where multiple tenants can send and receive their own events in real-time, with strict tenant isolation.

What to Build

Backend Service (Node.js/Express or Python/FastAPI):

- WebSocket server that handles connections with tenant authentication
- Simple in-memory event storage per tenant
- REST endpoint to post events: `POST /events` (with tenant header)
- WebSocket broadcasts events only to same-tenant connections

Frontend (Simple HTML/JS):

- Basic page with tenant login (dropdown: "Tenant A" or "Tenant B")
- Real-time event list that updates via WebSocket
- Form to send new events
- Clear visual indication when events arrive

Data Model:

```
{
  "id": "uuid",
  "tenant_id": "tenant_a",
  "message": "User logged in",
  "timestamp": "2025-01-20T10:30:00Z"
}
```

Requirements

- **Tenant Isolation:** Tenant A never sees Tenant B's events
- **Real-time:** Events appear in UI within 1 second
- **Simple Auth:** Use tenant ID in header/query param (no complex JWT)
- **In-Memory Only:** No database required

Success Criteria

- Two browser windows (different tenants) show different event streams
- Posting event from one tenant appears only in that tenant's windows
- Clean, documented code with setup instructions

Task 2: Multi-Tenant Document API

Problem

Create a secure document storage API where multiple tenants can upload, list, and download their files with role-based access control.

What to Build

Core API (Node.js/Express or Python/Flask):

- `POST /documents` - Upload file (form-data)
- `GET /documents` - List user's accessible documents
- `GET /documents/:id` - Download specific document
- `DELETE /documents/:id` - Delete document (admin only)

Authentication:

- Simple token-based auth with predefined users
- Users belong to tenants and have roles (admin/user)

Data Model:

```
{
  "id": "doc123",
  "tenant_id": "company_a",
  "filename": "contract.pdf",
  "uploaded_by": "user1",
  "upload_date": "2025-01-20",
  "access_level": "tenant" // "tenant" or "private"
}
```

Access Rules:

- Users see only their tenant's documents
- Regular users see "tenant" level docs + their own "private" docs
- Admins see all tenant documents

Requirements

- **Tenant Isolation:** Company A cannot access Company B's files
- **Role-Based Access:** Admin vs Regular user permissions
- **File Storage:** Local filesystem (./uploads/tenant_id/filename)
- **Security:** Input validation, no path traversal

Pre-configured Test Data

```
const USERS = {
  "admin_a": { tenant: "company_a", role: "admin", token: "token_admin_a" },
  "user_a": { tenant: "company_a", role: "user", token: "token_user_a" },
  "admin_b": { tenant: "company_b", role: "admin", token: "token_admin_b" }
};
```

Success Criteria

- Upload file as user_a, verify admin_a can access but admin_b cannot
- Admin can delete any tenant document, regular user cannot
- Clear error messages for unauthorized access

Task 3: Simple Fraud Detection API

Problem

Build a transaction analysis API that flags potentially fraudulent transactions using both rules and a simple ML model.

What to Build

Transaction Processing API:

- `POST /transactions` - Submit transaction for analysis
- `GET /transactions/flagged` - List recent flagged transactions
- Real-time processing with immediate fraud scoring

Detection Logic:

1. **Rules Engine:** Simple if/then rules (configurable thresholds)
2. **ML Component:** Use scikit-learn's IsolationForest for anomaly detection
3. **Scoring:** Combine rule flags + ML anomaly score

Transaction Model:

```
{
  "id": "tx123",
  "user_id": "user456",
  "amount": 1500.00,
  "location": "US",
  "timestamp": "2025-01-20T10:30:00Z",
  "merchant_category": "electronics"
}
```

Detection Rules:

- Amount > $5000 = High risk
- Same user, different countries within 1 hour = High risk
- Amount > 10x user's average = Medium risk

Requirements

- **ML Integration:** Train simple model on startup with synthetic data
- **Real-time:** Process transactions in <200ms
- **Configurable Rules:** Easy to adjust thresholds
- **Clear Scoring:** Return risk level (low/medium/high) with reasons

Simplified ML Approach

```
# Pre-generate training data on startup
training_data = generate_synthetic_transactions(1000)
model = IsolationForest(contamination=0.1)
model.fit(feature_matrix)

# For each transaction
ml_score = model.decision_function([transaction_features])[0]
rule_flags = check_rules(transaction)
final_risk = combine_scores(ml_score, rule_flags)
```

Success Criteria

- High-amount transaction gets flagged as "high risk"
- ML model detects outliers in transaction patterns
- API returns risk assessment in <500ms
- Include 5-10 test transactions with expected outcomes

Task 4 – AI-Powered Checkout & Upsell (≈ 1 working day)

Problem

Build a **tenant-aware e-commerce mini-workflow** that

1. offers catalogue, cart and checkout APIs,
2. calls an LLM at checkout to suggest up-sell items,
3. persists completed orders, and
4. runs both locally via Docker Compose and on any free-tier cloud.

What to Build

Component	Core Responsibilities	Minimum Scope
Catalog API	List products, prices and stock per tenant.	Read from in-memory list or JSON file; endpoints to list all products and fetch one by id.
Cart & Checkout API	Create/update cart; on checkout reserve stock and create order.	REST routes for cart creation/update and a checkout route that validates stock.
AI Upsell Service	When `UPSELL_ENABLED=true`, request ≤ 3 complementary products for the cart and add their explanations.	Decouple into its own module; may call OpenAI or a local model; must log prompt and response.
Order Store	Persist final orders.	In-memory map or SQLite file.
Public Storefront (SPA)	One-page UI: product list, “Add to cart”, Checkout button, upsell display, order confirmation.	Plain React, HTMX or vanilla JS.

Extra credit: feature flag `UPSELL_ENABLED`, and a promo code `SUMMER10` for 10 % discount.

Requirements

Area	Concrete Expectations
E-commerce logic	Stock checked atomically on checkout; calculate subtotal, 10 % promo discount (if present), then 20 % VAT.
Tenant isolation	All resources scoped by `tenant_id`; cross-tenant access must fail.
AI integration	Upsell logic isolated in its own file; checkout must succeed even if the LLM call fails or is disabled.
Quality gates	Linter + formatter; at least 5 unit tests and 1 integration test; GitHub Actions running tests.
Docs	Short OpenAPI or Postman spec; 2–4 diagram slides with context and sequence; README with local and cloud run instructions.
Deployability	`docker compose up` works locally; provide a live free-tier URL.

Success Criteria

- Checkout reserves stock, applies discount and VAT correctly, and stores the order.
- Upsell service returns relevant suggestions with reason and confidence when enabled.
- Toggle via `UPSELL_ENABLED` works without code changes.
- CI tests pass; code and docs are clean and idiomatic; both local and cloud deployments function.