TRADEFLOW OS

Production Build Plan

Python/FastAPI + React · Module-by-Module Architecture · User Stories & Acceptance Criteria

Production-grade system for a 10-person oil & gas trading company.
Python/FastAPI backend — native PDF parsing, first-class Claude SDK integration, Pydantic structured outputs. React frontend for rich interactivity. PostgreSQL with pgvector for AI-powered semantic search.
8 modules · 47 user stories · ~20 hours of active development with Claude Code.

BACKEND	Python 3.12 · FastAPI · SQLAlchemy 2.0 · Alembic · Pydantic v2 · Celery + Redis
FRONTEND	React 19 · Next.js 15 · TypeScript · Tailwind CSS · shadcn/ui · TanStack Query
DATABASE	PostgreSQL 16 + pgvector · Redis 7 · MinIO (S3-compatible)
AI LAYER	Anthropic Python SDK · pdfplumber · pgvector · Pydantic structured outputs
INFRA	Docker Compose · Nginx · GitHub Actions CI/CD · Uvicorn + Gunicorn
DEPLOY	Single VPS (Hetzner/DO) or AWS ECS · ~$50/mo for 10 users

TABLE OF CONTENTS

1. TECHNICAL ARCHITECTURE

Technical Architecture & Stack Decisions

Every technology choice is made for a 10-person trading company that needs reliability, performance, and maintainability. This system handles ~50-200 deals/month, ~10 concurrent users, and thousands of documents. The architecture is deliberately simple: no microservices, no Kubernetes.

Why Python/FastAPI for the Backend

Python is the optimal choice for this AI-heavy system:
1. Native PDF processing. pdfplumber, PyPDF2, pytesseract all run directly — no sidecar service, no inter-process communication, no extra Docker container. Upload → parse → structured data in one process.
2. First-class Claude integration. Anthropic's Python SDK is the most mature. Pydantic v2 structured outputs work seamlessly — define your schema as a Pydantic model, Claude returns validated, typed data. No JSON parsing gymnastics.
3. FastAPI IS Pydantic. Your request validation, response serialization, API documentation, and Claude structured outputs all use the SAME Pydantic models. Define once, use everywhere. This eliminates an entire category of type-drift bugs.
4. SQLAlchemy 2.0 async. Full async support with asyncpg. Concurrent document parsing, parallel Claude API calls, background jobs — all native. No callback hell, just async/await.
5. Claude Code generates Python fastest. The AI coding assistant produces higher-quality Python than any other language. Faster iteration, fewer bugs, less manual fixing.
6. Ecosystem depth. Every library you need exists and is battle-tested: Celery (background jobs), boto3 (S3), passlib (auth), python-jose (JWT), openpyxl (Excel export), reportlab (PDF generation).

Backend Stack Detail

Component	Choice	Why
Web Framework	FastAPI 0.115+	Async, auto-docs (OpenAPI), Pydantic-native, dependency injection, middleware ecosystem
ORM	SQLAlchemy 2.0 (async)	Mature, async with asyncpg, relationship loading, migration support via Alembic. Use mapped_column()
Migrations	Alembic	Auto-generates from SQLAlchemy models. Up/down migrations. Version-controlled schema.
Validation	Pydantic v2	Request/response schemas, Claude structured outputs, config management — all one library.
Auth	python-jose + passlib	JWT tokens (python-jose), bcrypt hashing (passlib). FastAPI Depends() for route protection.
Background Jobs	Celery 5.4 + Redis	Document parsing, embedding generation, payment reminders, notifications. Beat scheduler for cron job
ASGI Server	Uvicorn + Gunicorn	Uvicorn for async. Gunicorn as process manager (4 workers for 10 users).
File Storage	boto3 (S3/MinIO)	S3-compatible SDK. Works with MinIO locally, AWS S3 or DO Spaces in production.
AI / Claude	anthropic SDK	Official Python SDK. Structured outputs with Pydantic. Streaming support. Tool use.
PDF Parsing	pdfplumber + pytesseract	pdfplumber for digital PDFs. pytesseract OCR for scanned docs. Both native Python.
Embeddings	pgvector (asyncpg)	pgvector Python bindings. Store + query vector embeddings via SQLAlchemy.
API Docs	FastAPI auto-gen	Swagger UI + ReDoc auto-generated from Pydantic schemas. Zero config.
Testing	pytest + httpx	pytest-asyncio for async tests. httpx.AsyncClient for API testing. Factory Boy for test data.
Logging	structlog	Structured JSON logging. Request ID propagation. Async-safe.

Frontend Stack Detail

Component	Choice	Why
Framework	Next.js 15 (App Router)	File-based routing, SSR for initial load, React Server Components
Language	TypeScript (strict)	Type safety matching Pydantic backend schemas
Styling	Tailwind CSS v4	Utility-first. Fast iteration. No CSS-in-JS runtime cost.
Components	shadcn/ui	Copy-paste components. Full control. Dark theme native.
Data Tables	TanStack Table v8	Headless table — sorting, filtering, pagination. Critical for deal pipeline + proposal comparison.
Forms	React Hook Form + Zod	Performant forms. Zod schemas mirror Pydantic backend validation.
Data Fetching	TanStack Query v5	Server state with caching, background refetch, optimistic updates.
Charts	Recharts	React-native charting. Pipeline funnel, revenue, margin charts.
File Upload	react-dropzone	Drag-and-drop for RFQ/proposal document uploads.
State Mgmt	Zustand	Lightweight client state (UI prefs, sidebar). Server state via TanStack Query.

The Pydantic Advantage — One Schema Everywhere

This is the single biggest architectural advantage of Python for this project. In a Go or Node backend, you define types in 3+ places: database model, API request schema, API response schema, Claude prompt schema, frontend TypeScript type. With FastAPI + Pydantic:
1. SQLAlchemy model — defines the database table
2. Pydantic schema — defines API request validation, API response serialization, AND Claude structured output format
3. TypeScript type — generated from OpenAPI spec (one command: openapi-typescript)
Three definitions instead of six. When you change a field, you change it in one or two places, not five. For a trading system with 12 entities and dozens of fields each, this eliminates hundreds of potential type-drift bugs.

Infrastructure & Deployment

For 10 users, a single VPS with Docker Compose is the right architecture. No Kubernetes, no ECS, no multi-region. A $40-60/month server handles this with headroom to spare.

Component	Production Setup
Server	1x VPS: 4 vCPU, 8GB RAM, 160GB NVMe (Hetzner CPX31 ~$15/mo or DO $48/mo)
Database	PostgreSQL 16 on same VPS (managed DB if budget allows: DO ~$15/mo)
Object Storage	MinIO on VPS for MVP, migrate to S3/DO Spaces later (~$5/mo)
Redis	Redis 7 on same VPS — caching + Celery broker + session store
Reverse Proxy	Nginx with Let's Encrypt SSL auto-renewal (certbot)
ASGI	Gunicorn (4 workers) + Uvicorn workers. Systemd service or Docker.
CI/CD	GitHub Actions: lint → test → build Docker image → deploy via SSH

Monitoring	Uptime Robot (free) + structlog JSON → file. Sentry for error tracking (free tier).
Backups	pg_dump daily cron → S3/Spaces. Keep 30 days. Test restores monthly.
Domain	tradeflow.company.com — Cloudflare DNS (free tier) with proxy for DDoS

Build Order & Dependency Graph

With Claude Code, each module is one prompting session. Feed it the Pydantic models + user stories, let it generate FastAPI routers, service layers, SQLAlchemy models, and React pages. Each module takes 30-90 minutes. Total: ~20 hours for the full MVP.

#	Module	What It Does	Depends On	Time
M0	Foundation	Monorepo, Docker, all DB models, auth, RBAC, app shell	Nothing	~60m
M1	Deal Hub	Deal CRUD, status machine, pipeline view, deal detail page	M0	~45m
M3	Procurement	Vendors, proposals, comparison dashboard, vendor POs	M0, M1	~90m
M4	AI Engine	Document parsing, semantic search, discrepancy detection	M0, M1, M3	~90m
M2	CRM & Sales	Customers, quotes, customer PO intake, credit control	M0, M1	~60m
M5	Finance	AP/AR, milestone payments, invoicing, multi-currency P&L	M0, M3	~75m
M6	Dashboard	KPI cards, charts, activity feed, notifications	M0-M5	~45m
M7	Quality & Logistics	TPI, certs, freight, customs (Post-MVP)	M0, M1, M3	~90m

Critical Path

M0 → M1 → M3 → M4 → M2 → M5 → M6. Build Procurement (M3) before Sales (M2) because the proposal comparison dashboard is the most complex UI and the feature that sells the platform. M4 (AI) builds on M3's vendor/proposal data. Then Sales (M2) and Finance (M5) complete the money flow.

Realistic Timeline with Claude Code

Day	Modules	Hours	End State
Day 1	M0 + M1 + start M3	5-6h	Auth working, deals pipeline, vendor management started
Day 2	M3 + M4	5-6h	Full procurement, AI doc parsing, semantic vendor search
Day 3	M2 + M5	5-6h	Quotes, PO processing, payments, invoicing working
Day 4	M6 + polish + deploy	4-5h	Dashboard live, seed data, deployed to production
Day 5+	M7 + edge cases	Optional	Quality, logistics, advanced features

Project Structure

```
tradeflow-os/
■■■ backend/
■   ■■■ app/
■   ■   ■■■ main.py                  # FastAPI app factory, router registration
■   ■   ■■■ config.py                # Pydantic Settings (env-based config)
■   ■   ■■■ database.py              # Async SQLAlchemy engine + session
■   ■   ■■■ deps.py                  # FastAPI dependencies (get_db, get_current_user)
■   ■   ■■■ models/                  # SQLAlchemy ORM models
■   ■   ■   ■■■ __init__.py
■   ■   ■   ■■■ user.py
■   ■   ■   ■■■ deal.py
■   ■   ■   ■■■ customer.py
■   ■   ■   ■■■ vendor.py
■   ■   ■   ■■■ proposal.py
■   ■   ■   ■■■ quote.py
■   ■   ■   ■■■ customer_po.py
■   ■   ■   ■■■ vendor_po.py
■   ■   ■   ■■■ payment.py
■   ■   ■   ■■■ invoice.py
■   ■   ■   ■■■ document.py
■   ■   ■   ■■■ activity_log.py
■   ■   ■■■ schemas/                 # Pydantic request/response schemas
■   ■   ■   ■■■ deal.py              # DealCreate, DealResponse, DealUpdate
■   ■   ■   ■■■ customer.py
■   ■   ■   ■■■ vendor.py
■   ■   ■   ■■■ proposal.py
■   ■   ■   ■■■ quote.py
■   ■   ■   ■■■ payment.py
■   ■   ■   ■■■ invoice.py
■   ■   ■   ■■■ ai.py               # ParsedRFQ, ParsedProposal, Discrepancy
■   ■   ■■■ api/                     # FastAPI routers (HTTP handlers)
■   ■   ■   ■■■ auth.py
■   ■   ■   ■■■ deals.py
■   ■   ■   ■■■ customers.py
■   ■   ■   ■■■ vendors.py
■   ■   ■   ■■■ proposals.py
■   ■   ■   ■■■ quotes.py
■   ■   ■   ■■■ customer_pos.py
■   ■   ■   ■■■ vendor_pos.py
■   ■   ■   ■■■ payments.py
■   ■   ■   ■■■ invoices.py
■   ■   ■   ■■■ documents.py
■   ■   ■   ■■■ dashboard.py
■   ■   ■■■ services/                # Business logic layer
■   ■   ■   ■■■ deal_service.py
■   ■   ■   ■■■ ai_service.py        # Claude API: parse, compare, search
■   ■   ■   ■■■ embedding_service.py  # Vector embedding + pgvector search
■   ■   ■   ■■■ document_service.py   # Upload, parse (pdfplumber), index
■   ■   ■   ■■■ payment_service.py
■   ■   ■   ■■■ invoice_service.py
■   ■   ■   ■■■ fx_service.py         # Currency conversion
■   ■   ■■■ middleware/
■   ■   ■   ■■■ auth.py              # JWT validation
■   ■   ■   ■■■ rbac.py              # Role-based access control
■   ■   ■■■ workers/                 # Celery tasks
■   ■       ■■■ celery_app.py
■   ■       ■■■ parse_document.py
■   ■       ■■■ generate_embeddings.py
■   ■       ■■■ payment_reminders.py
■   ■■■ alembic/                     # Database migrations
■   ■   ■■■ alembic.ini
■   ■   ■■■ versions/
■   ■■■ seeds/                       # Demo data scripts
■   ■■■ tests/
■   ■■■ requirements.txt
■   ■■■ Dockerfile
■   ■■■ pyproject.toml
■■■ frontend/                        # Next.js 15 App
■   ■■■ app/
■   ■   ■■■ layout.tsx               # Shell + sidebar + auth provider
■   ■   ■■■ (auth)/login/page.tsx
■   ■   ■■■ dashboard/page.tsx
■   ■   ■■■ deals/
■   ■   ■   ■■■ page.tsx             # Pipeline (Kanban + table)
■   ■   ■   ■■■ new/page.tsx         # Create deal (+ AI RFQ parsing)
■   ■   ■   ■■■ [id]/page.tsx        # Deal detail (tabbed)
■   ■   ■■■ customers/
■   ■   ■■■ vendors/
■   ■   ■■■ finance/
■   ■■■ components/
■   ■   ■■■ deal-pipeline.tsx        # Kanban + table toggle
■   ■   ■■■ deal-chain.tsx           # Visual lifecycle chain
■   ■   ■■■ proposal-comparison.tsx  # Side-by-side dashboard [HERO]
■   ■   ■■■ quote-builder.tsx        # Margin calc + quote form
■   ■   ■■■ po-review.tsx            # PO vs Quote diff view
■   ■   ■■■ payment-schedule.tsx     # Timeline + actions
■   ■   ■■■ invoice-builder.tsx      # Invoice form + PDF
■   ■   ■■■ deal-pnl.tsx            # P&L; waterfall
■   ■■■ lib/
■       ■■■ api.ts                   # Typed API client (axios + interceptors)
■       ■■■ auth.ts                  # JWT handling + RBAC hooks
■       ■■■ types.ts                 # Auto-generated from OpenAPI spec
■■■ docker-compose.yml               # postgres, redis, minio, api, celery-worker, celery-beat,
 nextjs
■■■ Makefile
■■■ README.md
```

Foundation — Project Setup, Auth & Database

The base everything else builds on. One Claude Code session to scaffold the entire project.

M0-01 **As a developer**, I need a monorepo with Next.js frontend, FastAPI backend, PostgreSQL+pgvector, Redis, and MinIO running via Docker Compose. [MUST]

1. `docker-compose up` starts all services: postgres:16+pgvector, redis:7, minio, fastapi (uvicorn), celery-worker, celery-beat, nextjs
2. FastAPI on :8000 with auto-reload, Next.js on :3000 with HMR
3. PostgreSQL has pgvector extension enabled via init SQL script
4. MinIO on :9000 with default bucket 'documents'. Console on :9001.
5. Makefile: `make dev`, `make migrate`, `make seed`, `make test`, `make generate-types`

M0-02 **As a developer**, I need all 12 SQLAlchemy models with Alembic migrations, plus Pydantic schemas for every entity. [MUST]

1. 12 models: User, Customer, Vendor, Deal, VendorProposal, Quote, CustomerPO, VendorPO, Payment, Invoice, Document, ActivityLog
2. All IDs are UUIDs (uuid7 for sortability). All timestamps UTC (DateTime with timezone=True).
3. JSONB columns for: contacts, line_items, payment_schedule, parsed_data, discrepancies, certifications
4. pgvector column: Document.embedding = mapped_column(Vector(1536), nullable=True)
5. Indexes on: Deal(status, customer_id), VendorProposal(deal_id), Payment(status, due_date)
6. Pydantic schemas: Create, Update, Response variants for each entity. Response schemas use model_config = ConfigDict(from_attributes=True)
7. Alembic auto-generates initial migration from all models

M0-03 **As a user**, I can log in with email/password and see role-appropriate navigation. RBAC enforced on every API endpoint. [MUST]

1. POST /auth/login returns JWT (httpOnly cookie, 24h expiry) + refresh token
2. FastAPI Depends(): get_current_user (JWT validation), require_role('admin', 'sales') (RBAC)
3. 7 roles: admin, sales, procurement, finance, quality, logistics, warehouse
4. Password: passlib bcrypt. Tokens: python-jose. Cookies: httpOnly, SameSite=Lax, Secure in prod.
5. Frontend: useAuth() hook → user, role, isAuthenticated. Protected routes redirect to /login.
6. Sidebar visibility per role. Sales sees deals+customers+quotes. Procurement sees deals+vendors+proposals.

M0-04 **As a user**, I see a professional app shell — collapsible sidebar, top bar, dark theme, toast notifications, loading skeletons. [MUST]

1. Collapsible sidebar with icon + label nav. Top bar: breadcrumbs, notification bell, user dropdown.
2. Dark theme by default (shadcn/ui dark). Toast system (sonner). Error boundaries. Loading skeletons.
3. Responsive: sidebar collapses to icons on mobile.

M0-05 **As a developer**, I need typed API client (TypeScript) auto-generated from FastAPI's OpenAPI spec, with TanStack Query hooks. [MUST]

1. Run `openapi-typescript http://localhost:8000/openapi.json -o lib/types.ts` to generate types
2. Axios API client: JWT cookie handling, error interceptors (401→login, 403→toast, 500→toast)
3. TanStack Query hooks: useDeals(), useDeal(id), useCreateDeal(), etc. with cache invalidation

Deal Hub — The Central Nervous System

Every module connects to deals. The Deal is the single source of truth for every transaction in the system.

M1-01 **As a sales user**, I can create a new deal by entering customer, RFQ reference, description, currency, and dynamic line items. [MUST]

1. POST /api/deals creates deal with auto-generated deal_number (TF-2026-0001)
2. Form: searchable customer dropdown, RFQ ref, description, currency (AED/USD/EUR)
3. Dynamic line items: add/remove rows {description, material_spec, quantity, unit, required_delivery_date}
4. Starts in 'rfq_received' status. Activity log records creation with user + timestamp.

M1-02 **As a user**, I see all deals in a pipeline — toggleable between Kanban board (by status) and sortable/filterable table. [MUST]

1. GET /api/deals with query params: status, customer_id, value range, date range, sort, pagination
2. Kanban: columns per status, cards show deal#, customer, value, days-in-stage
3. Table: Deal#, Customer, Description, Value, Status, Margin%, Created. Sortable + searchable.
4. Filter bar: status multi-select, customer dropdown, date range. 'New Deal' button.

M1-03 **As a user**, I view a deal detail page with tabs: Overview, Vendor Proposals, Documents, Payments, Activity Log. [MUST]

1. GET /api/deals/{id} returns deal with all nested entities (proposals, quotes, POs, payments, docs)
2. Overview: status badge + change, customer info, line items, deal chain visualization, margin
3. Proposals tab: list with status badges, 'Find Vendors' (M3), comparison link
4. Documents: files with type badges, upload, preview/download. Payments: AP+AR timeline (M5).
5. Activity: chronological feed from ActivityLog — who did what, when, old→new values.

M1-04 **As a user**, I see a visual deal chain: RFQ → Quote → Customer PO → Vendor PO → Delivery → Invoice → Payment. [SHOULD]

1. Horizontal connected nodes on overview. Each: entity + status (green/yellow/grey).
2. Click node → jump to section. Shows linked entities: 'Quote Q-2026-0012 v2 (Sent)'.
3. Margin indicator between customer and vendor values.

M1-05 **As admin**, the system enforces valid deal status transitions. [MUST]

1. State machine: rfq_received → sourcing → quoted → po_received → ordered → in_production → shipped → delivered → invoiced → paid → closed
2. Back-transitions allowed: quoted → sourcing (revise), po_received → quoted (amend). Cancel from any pre-delivery state.
3. PATCH /api/deals/{id}/status validates transition + records in activity log.

M1-06 **As a user**, every deal change auto-logs: who, when, what field, old value, new value. [MUST]

1. ActivityLog created on: creation, status change, field update, doc upload, payment event
2. GET /api/deals/{id}/activities — paginated reverse-chronological feed

CRM & Sales — Customers, Quotes & Customer POs

The sell-side. Customer management, quote building with margins, purchase order processing.

M2-01 **As a sales user**, I manage customers — create, edit, search, view profile with deal history and credit info. [MUST]

1. Customer list: searchable table (name, type, credit limit, outstanding, # deals)
2. Detail: profile, credit summary (limit vs used), contacts, deal history table, notes
3. Create/edit: name, type, credit_limit, payment_terms_days, currency, contacts (dynamic), tax_id

M2-02 **As a sales user**, I build customer quotations — line items, margin calculator, terms — with version tracking. [MUST]

1. 'Create Quote' from deal. Pre-fills line items. Set unit_price per line — margin auto-calculates.
2. Highlights below 10% threshold. Fields: validity, Incoterms, payment terms. Versioning (v1, v2...).
3. Status: draft → sent → revised → accepted → expired → rejected. Quote list shows all versions.

M2-03 **As a sales user**, I record a customer PO and system flags discrepancies vs accepted quote. [MUST]

1. 'Record Customer PO' after quote accepted. Upload doc + enter details. AI parsing (M4) pre-fills.
2. Auto-comparison: price, quantity, delivery, terms. Diff view with severity (critical/minor).
3. 'Accept PO' or 'Request Amendment'. Accepting advances deal to 'po_received'.

M2-04 **As a sales user**, I track quote follow-ups — pending, expiring, needs revision. [SHOULD]

1. Quote list across deals. Expiring in 7d = yellow, expired = red. Quick actions: sent, accepted, revise.

M2-05 **As a finance user**, I see customer credit exposure and can place customers on credit hold. [SHOULD]

1. Credit summary: limit, used, available, utilization %. Warning on new deals if near limit.
2. Credit hold flag blocks new quotes. Overview page: all customers by utilization %.

M2-06 **CHINESE WALL**: Sales users cannot see vendor cost data — only customer pricing and margin %. [MUST]

1. Sales API responses exclude vendor unit_price, total_price. Quote builder shows margin % only.
2. Procurement cannot access Quote endpoints (403). Admin/management see everything.

Procurement — Vendors, Proposals & Vendor POs

The buy-side and the heart of the business. The proposal comparison dashboard is the hero feature — spend extra time here.

M3-01 **As a procurement user**, I manage vendors — profile, certifications, credibility score, performance history. **[MUST]**

1. Vendor list: searchable. Credibility score color-coded (green>70, yellow>40, red<40).
2. Detail: certs, performance (on-time%, quality, lead time), transaction history, bank details.
3. Create/edit: name, country, certifications (tags), product_categories (tags), contacts.

M3-02 **As a procurement user**, from a deal I search for matching vendors — AI semantic search (M4) + keyword search. **[MUST]**

1. 'Find Vendors' opens search panel. AI semantic search on deal line items + keyword fallback.
2. Results: vendor, credibility, matching products, last transaction, historical price.
3. Multi-select → 'Request Proposals' creates VendorProposal records in 'requested' status.

M3-03 **As a procurement user**, I record vendor proposals — manually or via document upload with AI parsing. **[MUST]**

1. Per proposal → 'Record': unit_price, total, currency, lead_time, terms, validity, specs_match.
2. File upload → AI parse (M4) → pre-fill. Status changes to 'received'. Notes for discrepancies.

M3-04 **As a procurement user**, I compare all proposals side-by-side and select the best vendor. **[HERO FEATURE]** **[MUST]**

1. Comparison table: Vendor, Credibility, Unit Price, Total, Lead Time, Terms, Specs Match, Discrepancies
2. Color coding: best=green, worst=red per column. Sortable. Filters: specs_match, min credibility, max price.
3. 'Select Vendor' → status 'selected', others 'rejected'. AI discrepancy warnings (M4).
4. Historical price: last price from this vendor for similar items.

M3-05 **As a procurement user**, I create a back-to-back vendor PO with payment schedule after customer PO accepted. **[MUST]**

1. 'Place Vendor Order' pre-fills from proposal. Payment schedule: stages (Advance, Milestone, Balance) with amounts + dates.
2. Back-to-back link: same deal_id. Margin = CustomerPO.total - VendorPO.total. Advances deal to 'ordered'.

M3-06 **As a procurement user**, I track vendor PO production progress + milestone completions. **[SHOULD]**

1. Status: draft → sent → confirmed → in_production → completed. Production notes log.
2. Status changes trigger payment milestone checks.

M3-07 **As a procurement user**, vendor credibility auto-calculates from delivery, quality, pricing history. **[SHOULD]**

1. Score = on_time(30%) + quality(30%) + pricing_consistency(20%) + total_deals(10%) + recency(10%)
2. Auto-recalculate on deal close. Shown on list, detail, comparison dashboard.

AI Engine — Document Intelligence & Smart Search

The differentiator. Python makes this dramatically easier — native pdfplumber, Anthropic SDK, Pydantic structured outputs.

> **This module needs the most iteration.** AI prompts need tuning — expect 3-5 iterations on parsing prompts. Start with RFQ parsing (simplest), then proposals, then PO comparison. The huge Python advantage: your Pydantic response schemas work directly as Claude structured output schemas. Define once, validate everywhere.

M4-01 **As a sales user**, uploading an RFQ PDF auto-extracts line items, specs, quantities, dates, terms → pre-fills deal form. **[MUST]**

1. Upload PDF → pdfplumber extracts text (native Python, no sidecar!) → text sent to Claude with Pydantic schema
2. Schema (Pydantic model): ParsedRFQ with customer_name, rfq_ref, line_items[{description, material_spec, qty, unit, delivery_date}], terms
3. Claude structured output: `client.messages.create(response_format=ParsedRFQ)`
4. Async via Celery task. Frontend shows 'Parsing...' → pre-fills form. Fallback: manual entry.
5. Prompt includes O&G hints: API specs, ASTM standards, pipe grades, valve types. Temperature: 0.
6. pytesseract OCR fallback for scanned documents. Store raw text + parsed JSON on Document record.

M4-02 **As a procurement user**, uploading a vendor proposal auto-extracts pricing, lead time, specs, terms. **[MUST]**

1. Same pipeline, different Pydantic schema: ParsedProposal (vendor_name, unit_price, total, currency, lead_time, terms, validity, certs)
2. Handles varied formats. Pre-fills VendorProposal form fields.

M4-03 **As a procurement user**, 'Find Vendors' uses semantic search on transaction history. **[MUST]**

1. Embed deal line items text via embedding model → store in pgvector column on Document
2. Also embed vendor product_categories + past deal descriptions on creation/update
3. Query: deal text → cosine similarity on pgvector → rank by similarity(0.6) x credibility(0.3) x recency(0.1)
4. SQLAlchemy query with pgvector: `Document.embedding.cosine_distance(query_embedding)`
5. Fallback to keyword search if no semantic matches above threshold

M4-04 **As a procurement user**, system auto-flags proposal discrepancies — spec mismatches, pricing outliers, missing info. **[SHOULD]**

1. Compare each proposal vs deal line items via Claude. Flag: spec mismatches, missing items, qty diffs.
2. Statistical: price >2σ from mean. Lead time vs required delivery. Output: [{proposal_id, field, severity, desc}].
3. Warnings shown on comparison dashboard (M3-04) with icons.

M4-05 **As a sales user**, customer PO upload auto-compares vs accepted quote, generates discrepancy report. **[MUST]**

1. Parse PO → compare vs Quote fields: line items (qty, price), total, delivery, terms.
2. Output: [{field, quote_value, po_value, match, severity (critical/warning/info)}]
3. Stored on CustomerPO.discrepancies_vs_quote. Displayed as diff view on PO review page.

M4-06 **As a user**, all documents are searchable by content, type, deal, date. **[SHOULD]**

1. GET /api/documents with filters. Full-text search on parsed_data (PostgreSQL tsvector or ILIKE).
2. Document list: type badge, deal#, uploaded by, date, parsed status. Preview + download.

8. MODULE 5 — FINANCE

Finance — Payments, Invoicing & Deal P&L

Track every dirham in and out, across currencies, with deal-level profitability.

M5-01 **As a finance user**, I see all payments (AP+AR) across all deals in one unified view. [MUST]

1. Finance page: tabs All, Payables (AP), Receivables (AR), Overdue
2. Table: Deal#, Direction, Type, Counterparty, Amount, Currency, Due Date, Status, Days Overdue
3. Summary cards: Total AP pending, Total AR pending, Total overdue, Net position
4. Quick actions: 'Mark as Paid' (date + ref) or 'Approve Payment'

M5-02 **As a finance user**, I process vendor payments per schedule — approve, record, track FX impact. [MUST]

1. Payment schedule on deal page + VendorPO detail. Workflow: scheduled → pending_approval → approved → paid.
2. Records approver + timestamp. On 'Mark as Paid': paid_date, payment_ref, actual_amount, fx_rate.
3. FX rate captured at payment time for margin impact calculation.

M5-03 **As a finance user**, I create invoices (proforma/final) linked to deals, with VAT and payment terms. [MUST]

1. 'Create Invoice' from deal. Pre-fills from quote. Type: proforma or final.
2. Auto-number INV-YYYY-NNNN. VAT: 0% or 5%. PDF generation (reportlab).
3. Status: draft/sent/partially_paid/paid/overdue. Auto-creates Payment record (AR).

M5-04 **As a finance user**, I track collections — aging buckets, partial payments, reminders. [MUST]

1. Aging: Current, 1-30d, 31-60d, 61-90d, 90+. Partial payments supported.
2. Reminder flags + escalation levels. Manual 'Send Reminder' action logs to activity.

M5-05 **As a finance user**, all payments support multi-currency (AED/USD/EUR/GBP/CNY) with FX rate capture. [MUST]

1. Every Payment: amount, currency, fx_rate (to AED), amount_base_currency. Totals always in AED.
2. FX API (exchangerate-api.com free tier) or manual rate entry.

M5-06 **As a manager**, I see per-deal P&L — revenue minus all costs, estimated vs actual margin. [MUST]

1. Deal P&L card: Revenue (CustomerPO) - Costs (VendorPO + logistics). All in AED at actual FX rates.
2. Comparison: estimated_margin_pct (at quote time) vs actual_margin_pct. Green/red indicator.

9. MODULE 6 — DASHBOARD

Dashboard & Reporting

What the business owner opens every morning. Quick build — aggregation queries + Recharts.

M6-01 **As a manager**, KPI cards: active deals, pipeline value, outstanding AR, overdue payments, avg margin, closed this month. [MUST]

1. 6 KPI cards with trend indicators (vs previous period). Data from aggregation queries.

M6-02 **As a manager**, charts: pipeline funnel, revenue by month, margin trend, top customers. [SHOULD]

1. Pipeline funnel (bar by status), Revenue by month (stacked bar), Margin trend (line), Top customers (bar).

M6-03 **As a user**, recent activity feed — last 20 events filtered by my RBAC access. [SHOULD]

1. From ActivityLog. User avatar, action, deal ref, relative timestamp. Click → deal.

M6-04 **As a user**, notification center: payments due, quotes expiring, PO discrepancies, overdue invoices. [NICE]

1. Bell icon + unread count. Generated by Celery Beat scheduled task (daily). Click → entity.

10. MODULE 7 — POST-MVP

Quality, Compliance & Logistics

Build after core MVP is live and being used. Important but doesn't block initial value delivery.

ID	Story	Priority
M7-01	Schedule/track Third-Party Inspections (TPI) for vendor POs	SHOULD
M7-02	Upload/track material certificates (MTRs, CoC) with AI parsing	SHOULD
M7-03	Build customer documentation packages — assemble certs against checklist	SHOULD
M7-04	Create/track Non-Conformance Reports (NCRs) with corrective actions	NICE
M7-05	Collect and compare shipping quotes from freight forwarders	SHOULD
M7-06	Track shipments in transit, allocate freight costs to deal P&L	SHOULD
M7-07	Customs clearance docs (HS codes, CoO) with free zone compliance	NICE
M7-08	Goods receipt (GRN), partial rejections, quality disputes workflow	SHOULD

Production Hardening Checklist

For a 10-person company handling real money and contracts, these ship from Day 1 — not 'later'.

Category	Item	Implementation
Security	**Password hashing**	passlib + bcrypt, cost=12
Security	**JWT in httpOnly cookies**	SameSite=Lax, Secure=True in prod. NOT localStorage.
Security	**CORS**	CORSMiddleware: allow only frontend origin. No wildcard.
Security	**Rate limiting**	slowapi: 20 req/min for auth, 200 req/min for API per IP.
Security	**Input validation**	Pydantic v2 on ALL request bodies. Field validators for business rules.
Security	**SQL injection**	SQLAlchemy parameterized queries only. Never raw f-string SQL.
Security	**File upload**	Validate MIME (python-magic), max 50MB, virus scan optional. Store in MinIO.
Security	**Secrets management**	python-dotenv for dev, env vars in prod. Never commit .env files.
Reliability	**Graceful shutdown**	Uvicorn signal handling. Drain connections on SIGTERM.
Reliability	**Health checks**	/healthz (app), /readyz (app+DB+Redis). Docker HEALTHCHECK directive.
Reliability	**Connection pooling**	SQLAlchemy async: pool_size=10, max_overflow=5, pool_recycle=3600.
Reliability	**Error handling**	FastAPI exception handlers. Never expose tracebacks. Structured error JSON.
Reliability	**DB backups**	pg_dump daily cron → S3. Retain 30 days. Test restore monthly.
Reliability	**Migration safety**	Always add columns nullable first. Never drop in same release. Test down().
Observability	**Structured logging**	structlog with JSON. Request ID in every log line. Log all auth events.
Observability	**Request tracing**	Middleware: log method, path, status, duration, user_id per request.
Observability	**Error tracking**	Sentry (free tier): captures unhandled exceptions with context.
Observability	**Uptime**	Uptime Robot (free): ping /healthz every 5 min. Alert on Slack/email.
Data	**Audit trail**	ActivityLog for ALL mutations. Soft delete: deleted_at column, never hard delete.
Data	**Transactions**	async with session.begin(): for multi-table ops (deal creation, payment processing).

Complete Data Model Reference

All 12 tables with column definitions. Copy this section directly into Claude Code when scaffolding M0. These become your SQLAlchemy models AND Pydantic schemas.

User

```
id UUID PK default uuid7(), email TEXT UNIQUE NOT NULL, password_hash TEXT NOT NULL, full_name TEXT NOT NULL,
role ENUM(admin/sales/procurement/finance/quality/logistics/warehouse) NOT NULL, is_active BOOL DEFAULT true,
created_at TIMESTAMPTZ DEFAULT now()
```

Customer

```
id UUID PK, name TEXT NOT NULL, type ENUM(operator/epc/contractor/other), credit_limit DECIMAL(15,2),
credit_used DECIMAL(15,2) DEFAULT 0, payment_terms_days INT DEFAULT 30, currency TEXT DEFAULT 'AED', contacts
JSONB [{name, title, email, phone}], address TEXT, tax_id TEXT, notes TEXT, is_on_hold BOOL DEFAULT false,
created_at TIMESTAMPTZ, updated_at TIMESTAMPTZ
```

Vendor

```
id UUID PK, name TEXT NOT NULL, country TEXT, certifications JSONB [strings], product_categories JSONB
[strings], credibility_score INT DEFAULT 50 CHECK(0-100), avg_lead_time_days INT, on_time_rate DECIMAL(3,2)
CHECK(0-1), quality_score INT CHECK(0-100), payment_terms TEXT, contacts JSONB, bank_details JSONB {bank,
account, swift, iban}, notes TEXT, created_at TIMESTAMPTZ, updated_at TIMESTAMPTZ
```

Deal

```
id UUID PK, deal_number TEXT UNIQUE (auto: TF-YYYY-NNNN), status ENUM(rfq_received/sourcing/quoted/po_receive
d/ordered/in_production/shipped/delivered/invoiced/paid/closed/cancelled), customer_id UUID FK→Customer NOT
NULL, customer_rfq_ref TEXT, description TEXT, line_items JSONB [{description, material_spec, quantity, unit,
required_delivery_date}], estimated_margin_pct DECIMAL(5,2), actual_margin_pct DECIMAL(5,2), currency TEXT
DEFAULT 'AED', total_value DECIMAL(15,2), total_cost DECIMAL(15,2), notes TEXT, created_by UUID FK→User,
created_at TIMESTAMPTZ, updated_at TIMESTAMPTZ
```

VendorProposal

```
id UUID PK, deal_id UUID FK→Deal NOT NULL, vendor_id UUID FK→Vendor NOT NULL, status
ENUM(requested/received/selected/rejected), unit_price DECIMAL(15,4), total_price DECIMAL(15,2), currency
TEXT, lead_time_days INT, payment_terms TEXT, validity_date DATE, specs_match BOOL, discrepancies JSONB, notes
TEXT, raw_document_url TEXT, parsed_data JSONB, created_at TIMESTAMPTZ
```

Quote

```
id UUID PK, deal_id UUID FK→Deal NOT NULL, quote_number TEXT UNIQUE (auto: Q-YYYY-NNNN), version INT DEFAULT
1, status ENUM(draft/sent/revised/accepted/expired/rejected), line_items JSONB [{description, quantity, unit,
unit_price, total}], subtotal DECIMAL(15,2), margin_pct DECIMAL(5,2), margin_amount DECIMAL(15,2), total_price
DECIMAL(15,2), currency TEXT, validity_date DATE, delivery_terms TEXT (Incoterms), payment_terms TEXT, notes
TEXT, sent_at TIMESTAMPTZ, created_at TIMESTAMPTZ
```

CustomerPO

```
id UUID PK, deal_id UUID FK→Deal NOT NULL, customer_id UUID FK→Customer NOT NULL, po_number TEXT NOT NULL,
status ENUM(received/under_review/accepted/amended/rejected), total_value DECIMAL(15,2), currency TEXT,
delivery_date DATE, payment_terms TEXT, discrepancies_vs_quote JSONB [{field, quote_value, po_value,
severity}], raw_document_url TEXT, parsed_data JSONB, created_at TIMESTAMPTZ
```

VendorPO

```
id UUID PK, deal_id UUID FK→Deal NOT NULL, vendor_id UUID FK→Vendor NOT NULL, po_number TEXT UNIQUE (auto:
VP-YYYY-NNNN), status ENUM(draft/sent/confirmed/in_production/completed/cancelled), line_items JSONB,
total_cost DECIMAL(15,2), currency TEXT, payment_schedule JSONB [{stage, label, amount, due_date, status,
paid_date}], expected_delivery DATE, notes TEXT, created_at TIMESTAMPTZ
```

Payment

```
id UUID PK, deal_id UUID FK→Deal NOT NULL, direction ENUM(outbound/inbound), type
ENUM(advance/milestone/balance/final), counterparty_type ENUM(vendor/customer), counterparty_id UUID,
reference_type ENUM(vendor_po/invoice), reference_id UUID, amount DECIMAL(15,2) NOT NULL, currency TEXT NOT
NULL, fx_rate DECIMAL(10,6) DEFAULT 1.0, amount_base_currency DECIMAL(15,2), status
ENUM(scheduled/pending_approval/approved/paid/overdue/cancelled), due_date DATE NOT NULL, paid_date DATE,
payment_ref TEXT, notes TEXT, approved_by UUID FK→User, created_at TIMESTAMPTZ
```

Invoice

`id UUID PK, deal_id UUID FK→Deal NOT NULL, customer_id UUID FK→Customer NOT NULL, invoice_number TEXT UNIQUE (auto: INV-YYYY-NNNN), type ENUM(proforma/final), status ENUM(draft/sent/partially_paid/paid/overdue/cancelled), line_items JSONB, subtotal DECIMAL(15,2), vat_rate DECIMAL(4,2) DEFAULT 0, vat_amount DECIMAL(15,2) DEFAULT 0, total DECIMAL(15,2), currency TEXT, payment_terms_days INT, due_date DATE, issued_at TIMESTAMPTZ, paid_amount DECIMAL(15,2) DEFAULT 0, notes TEXT, created_at TIMESTAMPTZ`

Document

`id UUID PK, deal_id UUID FK→Deal (nullable), entity_type ENUM(rfq/proposal/customer_po/vendor_po/invoice/certificate/mtr/inspection/other), entity_id UUID (nullable), file_url TEXT NOT NULL, file_name TEXT NOT NULL, file_size BIGINT, mime_type TEXT, uploaded_by UUID FK→User NOT NULL, parsed BOOL DEFAULT false, parsed_data JSONB (nullable), embedding VECTOR(1536) (nullable, pgvector), created_at TIMESTAMPTZ`

ActivityLog

`id UUID PK, deal_id UUID FK→Deal (nullable), user_id UUID FK→User NOT NULL, entity_type TEXT NOT NULL, entity_id UUID NOT NULL, action ENUM(created/updated/status_changed/uploaded/deleted), changes JSONB [{field, old_value, new_value}], created_at TIMESTAMPTZ DEFAULT now()`

End of Document

TradeFlow OS Build Plan · Python/FastAPI + React · Production Grade