alsakr online - AnythingLLM IDE Implementation Prompt Series
📋 Overview
This is a 6-phase implementation plan for building alsakr online using AnythingLLM IDE. Each phase is a separate, executable prompt that builds on the previous one.

🎯 PHASE 1: Project Foundation & Core Infrastructure
PROJECT CONTEXT:
You are building "alsakr online" - an AI-powered online spare parts marketplace for the MENA region. This is Phase 1: Foundation Setup.

TECHNICAL REQUIREMENTS:
- Language: Python 3.11+
- Framework: FastAPI for backend API
- Database: SQLite with proper schema design
- Project structure: Modular, production-ready architecture

YOUR TASK:
Create the complete project foundation with the following structure:

1. PROJECT DIRECTORY STRUCTURE:
```
alsakr-online/
├── backend/
│   ├── app/
│   │   ├── __init__.py
│   │   ├── main.py (FastAPI app initialization)
│   │   ├── config.py (environment & settings)
│   │   ├── database.py (SQLite connection handler)
│   │   └── models/
│   │       ├── __init__.py
│   │       ├── user.py (User model with bilingual support)
│   │       ├── part.py (online part catalog model)
│   │       ├── inquiry.py (RFQ/inquiry tracking model)
│   │       └── vendor.py (Vendor/supplier model)
│   ├── requirements.txt
│   └── .env.example
├── README.md
└── docker-compose.yml (optional, for future containerization)
```

2. DATABASE SCHEMA DESIGN:
Create SQLAlchemy models for:

a) Users table:
- id (UUID primary key)
- email, password_hash
- company_name, industry_type
- phone number
- preferred_language (en/ar)
- created_at, updated_at

b) Parts table:
- id (UUID primary key)
- part_number, manufacturer
- category, subcategory
- description_en, description_ar
- technical_specs (JSON field)
- image_url, datasheet_url
- status (active/discontinued)
- created_at, scraped_at

c) Inquiries table:
- id (UUID primary key)
- user_id (foreign key)
- part_id (foreign key)
- status (pending/quoted/closed)
- quantity, urgency_level
- notes, created_at

d) Vendors table:
- id (UUID primary key)
- company_name, contact_email
- country, response_rate
- avg_quote_time, reliability_score

3. FASTAPI APPLICATION:
Create main.py with:
- CORS middleware (for future frontend)
- Bilingual response handler (accept-language header)
- Health check endpoint: GET /api/health
- Basic auth endpoints: POST /api/auth/register, POST /api/auth/login
- Database initialization on startup

4. CONFIGURATION MANAGEMENT:
Create config.py with:
- Environment variable loading (python-decouple)
- Database URL configuration
- JWT secret key setup
- Supported languages list: ['en', 'ar']

5. REQUIREMENTS.txt:
Include these exact packages:
fastapi==0.104.1
uvicorn[standard]==0.24.0
sqlalchemy==2.0.23
python-decouple==3.8
python-jose[cryptography]==3.3.0
passlib[bcrypt]==1.7.4

python-multipart==0.0.6
pydantic==2.5.0
pydantic-settings==2.1.0

DELIVERABLES:
✅ Complete folder structure with all files
✅ Working FastAPI server that starts without errors
✅ Database models with proper relationships
✅ README.md with setup instructions
✅ .env.example with all required environment variables

VALIDATION:
The server should start with: `uvicorn backend.app.main:app --reload`
Health check should respond: GET http://localhost:8000/api/health

CODE QUALITY REQUIREMENTS:
- Type hints on all functions
- Docstrings for all classes and methods
- Proper error handling (try/except blocks)
- Bilingual field naming convention (field_en, field_ar)

Generate all files with complete, production-ready code. No placeholders or TODO comments.

🔍 PHASE 2: Web Scraping Engine (Scrub-Master Agent)
PROJECT CONTEXT:
You are implementing Phase 2 of alsakr online: The Scrub-Master Agent - a web scraping system that harvests online parts data from manufacturer websites (ABB, Siemens, Schneider Electric, SICK, Murrelektronik).

PREVIOUS PHASE:
✅ Phase 1 completed: FastAPI backend, database models, and core infrastructure are ready.

TECHNICAL REQUIREMENTS:
- Scraping framework: Playwright (headless browser automation)
- PDF processing: PyMuPDF (for datasheet extraction)
- Data storage: SQLite (using existing models from Phase 1)
- Rate limiting: Respect robots.txt, implement delays

YOUR TASK:
Create the complete web scraping system:

1. NEW DIRECTORY STRUCTURE:
backend/
├── app/
│   ├── scrapers/
│   │   ├── __init__.py

```
│   │   ├── base_scraper.py (Abstract base class)
│   │   ├── abb_scraper.py
│   │   ├── siemens_scraper.py
│   │   ├── schneider_scraper.py
│   │   ├── sick_scraper.py
│   │   └── murrelektronik_scraper.py
│   ├── utils/
│   │   ├── __init__.py
│   │   ├── pdf_extractor.py (Extract text from PDF datasheets)
│   │   ├── image_downloader.py (Save product images)
│   │   └── text_cleaner.py (Normalize part numbers, remove HTML)
│   └── api/
│       └── scraper_routes.py (Admin endpoints to trigger scraping)
```

2. BASE SCRAPER CLASS (base_scraper.py):
Create an abstract class with:
- __init__(self, brand_name, base_url, database_session)
- scrape_catalog() → main orchestration method
- extract_part_details(product_url) → individual part scraping
- save_to_database(part_data) → insert/update parts table
- handle_rate_limiting() → random delays (2-5 seconds)
- validate_part_number(part_num) → check format
- Error handling with retry logic (3 attempts)

3. MANUFACTURER-SPECIFIC SCRAPERS:
For EACH manufacturer, create a scraper that inherits from BaseScraper:

ABB Scraper (abb_scraper.py):
- Target: ABB product catalog pages
- Extract: Part number, product name, category, image, datasheet PDF link
- Handle: JavaScript-loaded content using Playwright
- Special logic: ABB uses product series numbering (e.g., 1SNA)

Siemens Scraper (siemens_scraper.py):
- Target: Siemens Mall or product pages
- Extract: MLFB part number, technical specs table, images
- Handle: Multi-page catalogs with pagination
- Special logic: Siemens has "successor parts" metadata

Schneider Electric Scraper (schneider_scraper.py):
- Target: Schneider Electric product database
- Extract: Reference number, product range, datasheet
- Handle: Country-specific catalogs (detect Egypt/MENA region)

SICK Scraper (sick_scraper.py):
- Target: SICK sensor catalog
- Extract: Part number, sensor type, specifications
- Handle: Technical specification tables

Murrelektronik Scraper (murrelektronik_scraper.py):
- Target: Murrelektronik product finder
- Extract: Article number, product family, datasheet

4. PDF DATASHEET EXTRACTOR (pdf_extractor.py):
Create functions:
- download_pdf(url, save_path) → download datasheet
- extract_text_from_pdf(pdf_path) → use PyMuPDF to get all text
- extract_specifications_table(pdf_text) → parse key-value pairs
- detect_successor_parts(pdf_text) → find replacement part mentions
- Return structured JSON: {
  "technical_specs": {...},
  "dimensions": {...},
  "successor_part": "..."
 }

5. ADMIN API ENDPOINTS (scraper_routes.py):
Create FastAPI routes:
- POST /api/admin/scraper/start/{brand} → Start scraping for specific brand
- GET /api/admin/scraper/status → Check scraping job status
- GET /api/admin/scraper/logs → View scraping logs
- POST /api/admin/scraper/stop → Emergency stop
- Require admin authentication (JWT token validation)

6. SCRAPING CONFIGURATION (scrapers/config.json):
Create a JSON config file:
```
{
  "abb": {
    "base_url": "https://new.abb.com/products",
    "catalog_url": "...",
    "selectors": {
      "product_card": ".product-item",
      "part_number": ".part-number",
      "datasheet_link": "a[href*='datasheet']"
    },
    "rate_limit_seconds": 3
  },
  // ... repeat for other brands
}
```

7. UPDATE requirements.txt:
Add these packages:
```
playwright==1.40.0
PyMuPDF==1.23.8
beautifulsoup4==4.12.2
aiohttp==3.9.1
fake-useragent==1.4.0
```

8. INITIALIZATION SCRIPT (scrapers/__init__.py):
Create a ScraperManager class:
- register_scraper(brand_name, scraper_class)
- get_scraper(brand_name) → return scraper instance
- run_all_scrapers() → sequential execution
- get_scraping_stats() → total parts scraped per brand

DELIVERABLES:
✅ 5 working manufacturer scrapers (ABB, Siemens, Schneider, SICK, Murrelektronik)
✅ PDF datasheet extraction pipeline
✅ Admin API to control scraping jobs
✅ Error handling and logging for failed scrapes
✅ Rate limiting to avoid IP bans

VALIDATION:
Run test scrape: POST http://localhost:8000/api/admin/scraper/start/abb
Verify: Check SQLite database for new part entries
Check: Datasheet PDFs saved in /backend/data/datasheets/

CODE QUALITY REQUIREMENTS:
- Use async/await for Playwright operations
- Implement proper exception handling (TimeoutError, NetworkError)
- Log all scraping activity to backend/logs/scraper.log
- Use Pydantic models for scraped data validation
- Comment complex CSS selectors and XPath expressions

IMPORTANT NOTES:
- Do NOT scrape all products at once (start with 50 parts per brand for testing)
- Respect robots.txt (check before implementing)
- Use rotating User-Agent headers to avoid detection
- Store raw HTML snapshots for debugging (optional)

Generate all files with complete, production-ready code. Include example usage in README.

🤖 PHASE 3: AI Multimodal Search Engine (Vision + Chat)
PROJECT CONTEXT:
You are implementing Phase 3 of alsakr online: The AI-Powered Multimodal Search Engine.
This system allows users to find online parts using:
1. Text descriptions (English/Arabic)
2. Image uploads (nameplate photos, worn parts)
3. Voice input (Arabic/English speech-to-text)

PREVIOUS PHASES:
✅ Phase 1: FastAPI backend + database models
✅ Phase 2: Web scraping system with 5 manufacturer scrapers

TECHNICAL REQUIREMENTS:
- Vision AI: CLIP model (OpenAI's CLIP-ViT-B-32)
- Language AI: Sentence Transformers (multilingual-e5-large)
- Vector Database: Qdrant (for semantic search)
- Voice AI: Faster-Whisper (for speech-to-text)
- LLM: Ollama with Llama 3.2 (for conversational queries)

YOUR TASK:
Create the complete AI search system:

1. NEW DIRECTORY STRUCTURE:
backend/
├── app/
│   ├── ai/
│   │   ├── __init__.py
│   │   ├── vision_agent.py (Image-based part identification)
│   │   ├── text_search.py (Semantic text search)
│   │   ├── voice_processor.py (Speech-to-text handler)
│   │   ├── embeddings.py (Vector embedding generation)
│   │   └── qdrant_client.py (Vector DB operations)
│   ├── api/
│   │   └── search_routes.py (Search API endpoints)
│   └── models/
│   └── search.py (Search request/response models)
├── data/
│   ├── images/ (uploaded user images)
│   ├── audio/ (uploaded voice notes)
│   └── embeddings/ (cached embeddings)

2. VECTOR DATABASE SETUP (qdrant_client.py):
Create QdrantManager class:
- initialize_collections() → Create 2 collections:
  * "parts_text" (for text embeddings, dim=1024)
  * "parts_images" (for image embeddings, dim=512)
- upsert_part_embedding(part_id, embedding, metadata)
- search_by_vector(query_embedding, top_k=10)
- filter_by_brand(brand_name) → Apply metadata filters
- get_part_by_id(part_id)

3. VISION AGENT (vision_agent.py):
Implement VisionAgent class:
- __init__() → Load CLIP model from transformers
- identify_part_from_image(image_path) → Main method:
  * Preprocess image (resize, normalize)
  * Extract CLIP image embedding (512-dim vector)
  * Search Qdrant "parts_images" collection
  * Return top 5 matching parts with similarity scores
- extract_text_from_image(image_path) → Use EasyOCR:

  * Extract visible text (part numbers, brand names)
  * Support Arabic + English text
  * Return cleaned text strings
- detect_nameplate_region(image_path) → Crop to nameplate:
  * Use simple object detection (optional, OpenCV)
  * Focus on text-dense regions

4. TEXT SEARCH ENGINE (text_search.py):
Create TextSearchEngine class:
- __init__() → Load multilingual-e5-large model
- search_by_description(query_text, language="en") → Main search:
  * Generate text embedding (1024-dim vector)
  * Search Qdrant "parts_text" collection
  * Apply language filter (match description_en or description_ar)
  * Return ranked results with scores
- search_by_part_number(part_num) → Exact match search:
  * Normalize part number (remove spaces, uppercase)
  * Query SQLite database directly
- hybrid_search(text_query, filters) → Combine:
  * Vector search + SQL filters (brand, category)
  * Re-rank results by relevance

5. VOICE PROCESSOR (voice_processor.py):
Implement VoiceProcessor class:
- __init__() → Load Faster-Whisper model ("base" size)
- transcribe_audio(audio_file_path, language="ar") → Main method:
  * Detect language automatically if not specified
  * Convert audio to text (supports Arabic dialects)
  * Return: {"text": "...", "language": "ar", "confidence": 0.95}
- process_voice_search(audio_file) → Complete pipeline:
  * Transcribe audio → Extract text
  * Pass text to TextSearchEngine.search_by_description()
  * Return search results

6. EMBEDDING GENERATOR (embeddings.py):
Create EmbeddingService class:
- generate_text_embedding(text) → Returns numpy array (1024-dim)
- generate_image_embedding(image_path) → Returns numpy array (512-dim)
- batch_embed_parts() → For initial database population:
  * Read all parts from SQLite
  * Generate embeddings for description_en + description_ar
  * Generate embeddings from image_url
  * Store in Qdrant collections
- update_single_part_embedding(part_id) → For new scraped parts

7. SEARCH API ENDPOINTS (search_routes.py):
Create FastAPI routes:

POST /api/search/text
- Body: {"query": "motor bearings high temperature", "language": "en"}
- Response: List of matching parts with scores

POST /api/search/image
- Body: Multipart form with image file
- Process: Save image → VisionAgent.identify_part_from_image()
- Response: Top 5 matching parts with similarity percentages

POST /api/search/voice
- Body: Multipart form with audio file (MP3, WAV)
- Process: Save audio → VoiceProcessor.transcribe_audio() → TextSearch
- Response: Transcription + search results

GET /api/search/part/{part_id}
- Response: Full part details + related parts

POST /api/search/advanced
- Body: Complex filters (brand, category, specs, price range)
- Process: Hybrid search (vector + SQL filters)

8. PYDANTIC MODELS (models/search.py):
Define request/response schemas:
- TextSearchRequest (query, language, filters)
- ImageSearchRequest (image file)
- VoiceSearchRequest (audio file, language)
- SearchResult (part_id, part_number, score, thumbnail)
- SearchResponse (results: List[SearchResult], query_time_ms, total_found)

9. OLLAMA INTEGRATION (Optional - Conversational Search):
Create chat_agent.py:
- query_llm(user_message, context_parts) → Ask Llama 3.2:
  * User: "I need a replacement for Siemens 6ES7 series PLC"
  * LLM: Generates search query + suggests filters
  * Call TextSearchEngine with generated query
- Bilingual system prompt (English + Arabic instructions)

10. UPDATE requirements.txt:
Add these packages:
qdrant-client==1.7.0
sentence-transformers==2.2.2
transformers==4.36.0
torch==2.1.0
pillow==10.1.0
easyocr==1.7.0
faster-whisper==0.10.0
opencv-python==4.8.1
numpy==1.24.3

11. INITIALIZATION SCRIPT:
Create backend/scripts/initialize_ai.py:
```python
# Run once to setup AI models and vector DB
# 1. Download all AI models
# 2. Create Qdrant collections
# 3. Generate embeddings for existing parts (from Phase 2 scraped data)
# 4. Verify search functionality with test queries
```

DELIVERABLES:
✅ Working image search (upload photo → get matching parts)
✅ Working text search (English + Arabic semantic search)
✅ Working voice search (Arabic speech → text → parts)
✅ Qdrant vector database with embeddings for all scraped parts
✅ API endpoints with proper error handling

VALIDATION TESTS:
1. Image Search Test:
   - Upload photo of ABB circuit breaker nameplate
   - Should return correct ABB part + similar alternatives

2. Text Search Test (English):
   - Query: "24V proximity sensor IP67 rated"
   - Should return SICK/Murrelektronik sensors

3. Text Search Test (Arabic):
   - Query: "محرك كهربائي سرعة عالية" (high-speed electric motor)
   - Should return relevant motors with Arabic descriptions

4. Voice Search Test:
   - Upload Arabic voice note: "أبحث عن حساس ضغط"
   - Should transcribe correctly and return pressure sensors

5. Performance Test:
   - Search response time < 2 seconds
   - Qdrant search on 10,000+ parts should be under 500ms

CODE QUALITY REQUIREMENTS:
- All AI model loading should be lazy (load on first use, not at startup)
- Implement proper memory management (clear GPU cache after inference)
- Use async operations for file uploads
- Add request validation (max file size: 10MB for images, 5MB for audio)
- Implement rate limiting (max 20 searches per minute per user)
- Log all search queries to backend/logs/search_queries.log for analytics

IMPORTANT NOTES:

- Download AI models during initialization, not at runtime
- Use GPU if available (CUDA), fallback to CPU
- Cache embeddings for frequently searched terms
- Implement search result caching (Redis in future phase)
- Store user search history for personalization (future feature)

Generate all files with complete, production-ready code. Include API usage examples in README.

🤝 PHASE 4: RFQ Automation & Vendor Management (Negotiator Agent)
PROJECT CONTEXT:
You are implementing Phase 4 of alsakr online: The Negotiator Agent - an automated system that sends RFQs (Request for Quotation) to vendors, aggregates responses, and manages the quote comparison workflow.

PREVIOUS PHASES:
✅ Phase 1: FastAPI backend + database
✅ Phase 2: Web scraping (5 manufacturers)
✅ Phase 3: AI multimodal search (vision/text/voice)

TECHNICAL REQUIREMENTS:
- Email automation: SMTP (Gmail) + IMAP (for reading responses)
- Workflow engine: n8n (self-hosted) OR custom Python scheduler
- PDF generation: ReportLab (for quote PDFs)
- Quote parsing: LLM-based (Ollama with Llama 3.2) to extract price/lead time from emails

YOUR TASK:
Create the complete RFQ automation system:

1. NEW DIRECTORY STRUCTURE:
backend/
├── app/
│ ├── rfq/
│ │ ├── __init__.py
│ │ ├── negotiator_agent.py (Main RFQ orchestrator)
│ │ ├── email_sender.py (SMTP email handler)
│ │ ├── email_parser.py (Parse vendor responses)
│ │ ├── quote_aggregator.py (Compare quotes)
│ │ └── pdf_generator.py (Generate quote comparison PDF)
│ ├── models/
│ │ ├── rfq.py (RFQ database model)
│ │ └── quote.py (Quote database model)
│ ├── api/
│ │ └── rfq_routes.py (RFQ API endpoints)
│ └── templates/
│ ├── email_rfq_en.html (English RFQ email template)
│ ├── email_rfq_ar.html (Arabic RFQ email template)
│ └── quote_comparison.html (HTML template for PDF)

2. DATABASE MODELS:
Add new SQLAlchemy models:

RFQ Model (models/rfq.py):
- id (UUID primary key)
- user_id (foreign key to users)
- part_id (foreign key to parts)
- quantity, urgency ("normal", "urgent", "emergency")
- target_price, max_lead_time_days
- status ("draft", "sent", "quoted", "closed")
- notes, created_at, sent_at
- Relationship: rfq.quotes (one-to-many)

Quote Model (models/quote.py):
- id (UUID primary key)
- rfq_id (foreign key to rfqs)
- vendor_id (foreign key to vendors)
- price_per_unit, total_price, currency
- lead_time_days, shipping_cost
- availability ("in_stock", "2_weeks", "4_weeks", "discontinued")
- quote_validity_days
- notes, raw_email_text
- status ("pending", "accepted", "rejected")
- created_at, expires_at

3. NEGOTIATOR AGENT (negotiator_agent.py):
Create NegotiatorAgent class with workflow:

Main method: process_rfq(rfq_id):
  Step 1: Load RFQ details from database
  Step 2: Find relevant vendors (query vendors table)
    - Filter by: country, part_category, reliability_score > 3.0
    - Select top 5 vendors
  Step 3: Generate personalized email for each vendor
    - Use bilingual templates (detect vendor language)
    - Include: Part details, quantity, urgency, delivery location
  Step 4: Send emails via EmailSender
  Step 5: Update RFQ status to "sent"
  Step 6: Schedule follow-up (check for responses after 48 hours)

Helper methods:
- select_vendors(part_id, quantity) → Returns List[Vendor]
- generate_rfq_email(vendor, rfq, language) → Returns HTML string
- estimate_shipping_cost(vendor_country, user_country, weight) → Returns float
- calculate_landed_cost(quote) → price + shipping + customs_estimate

4. EMAIL SENDER (email_sender.py):

Create EmailSender class:
- __init__(smtp_server, smtp_port, username, password)
- send_rfq_email(to_email, subject, html_body, attachments=None)
- send_bulk_rfqs(rfq_id, vendor_list) → Send to multiple vendors
- track_email_status(email_id) → Check if sent successfully
- handle_bounce_emails() → Mark vendor email as invalid

Email template variables:
- {{vendor_name}}, {{part_number}}, {{part_description}}
- {{quantity}}, {{urgency_badge}}, {{delivery_location}}
- {{user_company}}, {{user_contact}}, {{rfq_reference_number}}

5. EMAIL PARSER (email_parser.py):
Create EmailParser class using Ollama LLM:

Main method: parse_vendor_response(email_text, rfq_id):
  Step 1: Extract key information using LLM:
    - Prompt: "Extract: price per unit, lead time, availability, currency"
    - Use Llama 3.2 with structured output
  Step 2: Validate extracted data:
    - Price must be numeric and > 0
    - Lead time must be in days (convert from text: "2 weeks" → 14)
  Step 3: Create Quote object and save to database
  Step 4: Notify user via push notification (future: implement WebSocket)

LLM Prompt template:
You are parsing a vendor quote email. Extract the following:

Price per unit (numeric value)
Currency (USD, EUR, EGP, etc.)
Lead time in days
Availability status (in stock, 2 weeks, 4 weeks, discontinued)
Shipping cost (if mentioned)
Quote validity period

Email text:
{{email_content}}
Return JSON format:
{
"price_per_unit": 0.0,
"currency": "USD",
"lead_time_days": 0,
"availability": "in_stock",
"shipping_cost": 0.0,
"quote_validity_days": 30
}

6. QUOTE AGGREGATOR (quote_aggregator.py):

Create QuoteAggregator class:
- compare_quotes(rfq_id) → Returns sorted list of quotes:
  * Primary sort: Total landed cost (price + shipping + customs)
  * Secondary sort: Lead time (faster is better)
  * Tertiary sort: Vendor reliability_score
- generate_comparison_table(rfq_id) → HTML table for user dashboard
- recommend_best_quote(rfq_id, user_preferences) → AI-powered recommendation:
  * If user prefers speed: Prioritize lead time
  * If user prefers cost: Prioritize lowest price
  * If user prefers reliability: Prioritize vendor score
- flag_outlier_quotes(rfq_id) → Detect suspiciously low/high prices

7. PDF GENERATOR (pdf_generator.py):
Create QuotePDFGenerator class:
- generate_quote_comparison_pdf(rfq_id, output_path)
- Contents:
  * Header: Company logo, RFQ reference number
  * Part details: Image, part number, description, specs
  * Quote table: Vendor, Price, Lead Time, Total Cost
  * Recommendation section: "Best Value", "Fastest Delivery", "Most Reliable"
  * Footer: Terms & conditions, contact info
- Use ReportLab for PDF generation
- Support bilingual output (English/Arabic)

8. RFQ API ENDPOINTS (rfq_routes.py):
Create FastAPI routes:

POST /api/rfq/create
- Body: {part_id, quantity, urgency, notes}
- Response: Created RFQ object with rfq_id
- Auto-trigger: NegotiatorAgent.process_rfq(rfq_id)

GET /api/rfq/{rfq_id}
- Response: RFQ details + all associated quotes

GET /api/rfq/{rfq_id}/quotes
- Response: List of quotes sorted by best value

POST /api/rfq/{rfq_id}/accept-quote/{quote_id}
- Action: Mark quote as accepted, update RFQ status to "closed"
- Trigger: Send confirmation email to vendor

GET /api/rfq/user/history
- Response: User's past RFQs with status

POST /api/rfq/{rfq_id}/download-comparison
- Response: PDF file of quote comparison

9. VENDOR MANAGEMENT FEATURES:
Add to Vendor model (enhance Phase 1 model):
- email_template_language (en/ar)
- preferred_contact_method (email/whatsapp)
- response_time_hours (average)
- reliability_score (1-5 stars, calculated from past quotes)
- total_quotes_received, total_orders_fulfilled
- last_contacted_at

Create vendor_manager.py:
- update_vendor_score(vendor_id, rfq_id) → Recalculate after each interaction
- track_response_time(vendor_id, sent_at, responded_at)
- blacklist_vendor(vendor_id, reason) → Mark as unreliable

10. BACKGROUND TASKS:
Create tasks/rfq_scheduler.py using APScheduler:
- check_pending_rfqs() → Run every 6 hours:
  * Find RFQs with status "sent" and no quotes after 48 hours
  * Send reminder emails to vendors
  * Notify user if no responses received
- expire_old_quotes() → Run daily:
  * Mark quotes as expired if past validity date
  * Notify user to request new quotes
- sync_vendor_emails() → Run weekly:
  * Check for new vendor responses in inbox
  * Parse and create Quote objects

11. EMAIL MONITORING (IMAP):
Create email_monitor.py:
- connect_to_inbox(imap_server, username, password)
- fetch_unread_emails(filter="from known vendors")
- match_email_to_rfq(email_subject, email_body) → Use regex/LLM
- mark_as_processed(email_id)

12. UPDATE requirements.txt:
Add:
reportlab==4.0.7
apscheduler==3.10.4
python-dotenv==1.0.0
jinja2==3.1.2
requests==2.31.0

13. CONFIGURATION (config/rfq_config.py):
```python
RFQ_CONFIG = {
    "max_vendors_per_rfq": 5,
    "default_quote_validity_days": 30,
    "reminder_after_hours": 48,
```

```
    "max_retries_per_vendor": 2,
    "smtp": {
        "server": "smtp.gmail.com",
        "port": 587,
        "username": os.getenv("EMAIL_USERNAME"),
        "password": os.getenv("EMAIL_PASSWORD")
    },
    "imap": {
        "server": "imap.gmail.com",
        "port": 993
    }
}
```

DELIVERABLES:
✅ Automated RFQ creation and email sending to vendors
✅ LLM-based quote parsing from email responses
✅ Quote comparison dashboard with recommendations
✅ PDF generation for quote comparisons
✅ Background scheduler for follow-ups and reminders
✅ Vendor scoring and reliability tracking

VALIDATION TESTS:
1. RFQ Creation Test:
   - Create RFQ for ABB circuit breaker
   - Should send emails to 5 relevant vendors
   - Check email delivery status

2. Quote Parsing Test:
   - Manually create test vendor response email
   - EmailParser should extract price, lead time correctly
   - Quote should appear in database

3. Quote Comparison Test:
   - Create 3 mock quotes for same RFQ (different prices/lead times)
   - QuoteAggregator should rank them correctly
   - Best quoteContinue9:41 PMrecommendation should be logical

PDF Generation Test:

Generate comparison PDF for RFQ with 3 quotes
PDF should be bilingual (if user is Arabic)
Should include all required sections

CODE QUALITY REQUIREMENTS:

Implement retry logic for email sending (3 attempts with exponential backoff)
Use connection pooling for SMTP/IMAP
Log all sent emails to backend/logs/rfq_emails.log
Implement email queue system (don't send 100 emails at once)
Add email validation before sending (check email format, domain exists)
Handle email parsing failures gracefully (flag for manual review)

IMPORTANT NOTES:

Gmail SMTP has sending limits (500 emails/day for free accounts)
Use OAuth2 for Gmail authentication (more secure than app passwords)
Store email templates in Jinja2 format for easy customization
Implement email unsubscribe mechanism (required by law)
Add GDPR-compliant email consent tracking
Consider using Mailgun or SendGrid for production (higher limits)

Generate all files with complete, production-ready code. Include example email templates.

📊 PHASE 5: online CRM & User Dashboard
PROJECT CONTEXT:
You are implementing Phase 5 of alsakr online: The online Intelligence CRM - a specialized CRM that tracks machines, parts lifecycle, and procurement history (not just contact information).
PREVIOUS PHASES:
✅ Phase 1: Backend + database
✅ Phase 2: Web scraping
✅ Phase 3: AI search (vision/text/voice)
✅ Phase 4: RFQ automation
TECHNICAL REQUIREMENTS:

Backend: PocketBase (self-hosted, lightweight CRM backend)
Frontend: React + Next.js + Tailwind CSS + shadcn/ui
Real-time updates: WebSocket (PocketBase built-in)
Charts: Recharts (for analytics dashboards)

YOUR TASK:
Create the complete CRM and user dashboard:

NEW DIRECTORY STRUCTURE:
frontend/
├── src/
│   ├── app/
│   │   ├── (auth)/
│   │   │   ├── login/page.tsx
│   │   │   └── register/page.tsx
│   │   ├── (dashboard)/

```
│   │   │   ├── layout.tsx (Sidebar + Header)
│   │   │   ├── page.tsx (Main dashboard)
│   │   │   ├── fleet/page.tsx ("My Machines" asset management)
│   │   │   ├── inquiries/page.tsx (RFQ tracking board)
│   │   │   ├── quotes/page.tsx (Quote comparisons)
│   │   │   ├── history/page.tsx (Order history)
│   │   │   └── settings/page.tsx (User preferences)
│   │   └── layout.tsx (Root layout)
│   ├── components/
│   │   ├── ui/ (shadcn/ui components)
│   │   ├── dashboard/
│   │   │   ├── StatCard.tsx (KPI cards)
│   │   │   ├── InquiryKanban.tsx (Drag-drop board)
│   │   │   ├── QuoteComparison.tsx (Table with filters)
│   │   │   └── MachineCard.tsx (Asset card)
│   │   ├── search/
│   │   │   ├── ImageUpload.tsx (Drag-drop image)
│   │   │   ├── VoiceRecorder.tsx (Audio recording)
│   │   │   └── SearchResults.tsx (Grid of parts)
│   │   └── layout/
│   │       ├── Sidebar.tsx
│   │       ├── Header.tsx (with language toggle)
│   │       └── NotificationBell.tsx
│   ├── lib/
│   │   ├── api.ts (Axios API client)
│   │   ├── pocketbase.ts (PocketBase client)
│   │   └── i18n.ts (Internationalization: en/ar)
│   └── styles/
│       └── globals.css (Tailwind + custom RTL styles)
├── public/
│   ├── locales/
│   │   ├── en.json (English translations)
│   │   └── ar.json (Arabic translations)
├── package.json
├── next.config.js
└── tailwind.config.js

backend/
├── pocketbase/ (PocketBase binary + data)
│   ├── pb_data/ (SQLite database)
│   └── pb_migrations/ (Schema migrations)
```

POCKETBASE COLLECTIONS:
Define these collections in PocketBase:

machines (Asset Management):

user_id (relation to users)

machine_name, machine_type (Motor, PLC, Sensor, etc.)
manufacturer, model_number, serial_number
installation_date, location (Factory Hall 1, Line 3, etc.)
health_score (0-100, calculated from part age)
image_url, manual_url
last_maintenance_date, next_maintenance_due
parts (relation to parts - many-to-many)

part_history (Procurement tracking):

user_id (relation to users)
part_id (relation to parts)
machine_id (relation to machines)
action_type (searched, inquired, quoted, purchased)
quantity, price_paid, vendor_id
order_date, delivery_date
notes, invoice_url

notifications:

user_id (relation to users)
type (quote_received, price_drop, obsolescence_alert)
title, message, link
is_read, created_at

FRONTEND: MAIN DASHBOARD (dashboard/page.tsx):
Create a Bento Grid layout with these cards:

Top Row (KPIs):

Total Active Inquiries (count)
Quotes Received This Week (count)
Average Quote Response Time (hours)
Total Machines in Fleet (count)

Middle Row:

Recent Inquiries (table: Part, Status, Vendors Contacted, Last Update)
Quote Comparison Widget (show best 3 quotes for latest RFQ)

Bottom Row:

Recent Searches (list with images)
Maintenance Alerts (parts due for replacement based on lifecycle)

Use React Server Components + client components for interactivity
Implement real-time updates using PocketBase realtime subscriptions

FLEET MANAGEMENT PAGE (fleet/page.tsx):
Features:

Grid view of all machines (cards with images)
Each card shows:

Machine name, type, manufacturer
Health score (color-coded: green/yellow/red)
Last maintenance date
Quick action: "Search Parts for This Machine"

Add Machine button (opens modal form)
Click machine → Detail view:

Full specs
Parts history (table of all parts ever used)
Maintenance schedule (timeline)
Related documents (manuals, data sheets)

Create MachineCard.tsx component:

Health score visualization (progress bar or donut chart)
Status badge (Operational, Needs Maintenance, Critical)
Quick stats: Age, Parts Replaced, Last Service

INQUIRY TRACKING BOARD (inquiries/page.tsx):
Implement Kanban board with 4 columns:
Draft (RFQs not yet sent)
Sent (waiting for vendor responses)
Quoted (received at least 1 quote)
Closed (quote accepted or RFQ cancelled)

Each inquiry card shows:

Part image + name
Number of quotes received
Best quote price (if available)
Time since sent
Drag-and-drop to move between columns

Use @dnd-kit/core for drag-and-drop functionality
Filter options: By date range, by part category, by urgency

QUOTE COMPARISON PAGE (quotes/page.tsx):
Features:

Table with sortable columns:

Vendor Name, Price/Unit, Quantity, Total Cost
Lead Time, Shipping Cost, Landed Cost
Availability, Quote Validity
Reliability Score (vendor rating)

Highlight "Best Value" row (green background)
Action buttons: Accept Quote, Request Better Price, Download PDF
Side panel: Show vendor details (past performance, location, contact)

Create QuoteComparison.tsx component:

Use Recharts for price comparison bar chart
Calculate cost breakdown (pie chart: product cost, shipping, customs)
Show historical prices for this part (line chart)

SEARCH INTERFACE (components/search/):
Unified search component with 3 tabs:
Text Search:

Input field with autocomplete (suggest part numbers)
Language toggle (EN/AR)
Advanced filters (brand, category, price range)

Image Search:

Drag-and-drop zone
Live camera capture (mobile)
Preview with crop tool
Processing indicator (spinner + "Analyzing image...")

Voice Search:

Record button (animated while recording)
Audio waveform visualization
Transcription display (show recognized text)
Language detection indicator

Create SearchResults.tsx:

Grid of part cards (image, name, price range)
Quick action: "Request Quote" button
Save to favorites (heart icon)
Compare checkbox (select multiple parts)

NOTIFICATIONS SYSTEM:
Create NotificationBell.tsx component:

Badge with unread count
Dropdown with recent notifications (last 10)
Types:

"New quote received for ABB Circuit Breaker" (with price)
"Price drop alert: Siemens Motor now 15% cheaper"
"Part obsolescence: Schneider 12345 discontinued, view alternatives"
"Maintenance due: Motor #3 bearings need replacement in 7 days"

Mark all as read button
Link to full notifications page

Backend integration:

PocketBase realtime subscription to notifications collection
WebSocket updates (no page refresh needed)

USER SETTINGS PAGE (settings/page.tsx):
Sections:
Profile: Company name, industry, location
Preferences:

Default language (EN/AR)
Notification preferences (email, push, in-app)
Search defaults (preferred brands, max price range)

Integrations:

Connect WhatsApp Business (future feature placeholder)
Connect ERP system (API key input)

Billing: Subscription plan, payment method (future)
INTERNATIONALIZATION (lib/i18n.ts):
Implement with next-intl:

Detect browser language (default to Arabic for Egypt/MENA region)
RTL support for Arabic (flip layout, right-align text)
Number formatting (1,234.56 for EN, ١،٢٣٤٫٥٦ for AR)
Date formatting (different formats for EN/AR)
Currency display (EGP, USD, EUR)

Translation files (public/locales/):
en.json:
json{
  "dashboard": {
    "title": "Dashboard",
    "activeInquiries": "Active Inquiries",
    "quotesReceived": "Quotes Received"
  },
  "search": {
    "placeholder": "Search for parts...",
    "uploadImage": "Upload Image",
    "recordVoice": "Record Voice"
  }
}
ar.json (RTL):
json{
  "dashboard": {
    "title": "لوحة التحكم",
    "activeInquiries": "الاستفسارات النشطة",
    "quotesReceived": "العروض المستلمة"
  },
  "search": {
    "placeholder": "ابحث عن قطع الغيار...",
    "uploadImage": "تحميل صورة",
    "recordVoice": "تسجيل صوتي"
  }
}

ADMIN PANEL (separate section):
Create admin-only routes:

/admin/scrapers (manage scraping jobs)
/admin/vendors (approve new vendors, view scores)
/admin/users (user management, activity logs)
/admin/analytics (platform metrics, revenue dashboard)

Admin dashboard widgets:

Total parts in database (with growth chart)
Scraper status (running/failed jobs)
Active users (daily/weekly/monthly)
Revenue metrics (if monetized)
Top searched parts (word cloud or table)

RESPONSIVE DESIGN:
Mobile-first approach:

Hamburger menu for sidebar on mobile
Touch-friendly buttons (min 44px height)
Swipe gestures for Kanban board
Bottom navigation bar on mobile (Home, Search, Fleet, Profile)

Tablet optimization:

2-column layout for medium screens
Collapsible sidebar

Desktop:

Full Bento grid layout
Sidebar always visible
Multiple modals/popovers supported

PACKAGE.JSON:

```
json{
  "dependencies": {
    "next": "14.0.4",
    "react": "18.2.0",
    "react-dom": "18.2.0",
    "pocketbase": "0.20.0",
    "@dnd-kit/core": "6.1.0",
    "@dnd-kit/sortable": "8.0.0",
    "axios": "1.6.2",
    "recharts": "2.10.3",
    "next-intl": "3.4.0",
    "lucide-react": "0.294.0",
    "tailwindcss": "3.3.6",
    "@radix-ui/react-dialog": "1.0.5",
    "@radix-ui/react-dropdown-menu": "2.0.6",
```

```
    "class-variance-authority": "0.7.0",
    "clsx": "2.0.0",
    "tailwind-merge": "2.1.0"
  }
}
```

POCKETBASE SETUP SCRIPT:
Create backend/scripts/setup_pocketbase.sh:

```
bash#!/bin/bash
# Download PocketBase
wget
https://github.com/pocketbase/pocketbase/releases/download/v0.20.0/pocketbase_0.20.0_li
nux_amd64.zip
unzip pocketbase_0.20.0_linux_amd64.zip -d pocketbase/
cd pocketbase
./pocketbase serve --http=0.0.0.0:8090
```

DELIVERABLES:
✅ Complete React/Next.js frontend with all pages
✅ PocketBase CRM backend with all collections
✅ Real-time WebSocket notifications
✅ Bilingual UI (English + Arabic RTL)
✅ Responsive design (mobile/tablet/desktop)
✅ Kanban board for inquiry tracking
✅ Machine asset management interface
✅ Quote comparison with visualizations

VALIDATION TESTS:
1. Dashboard Load Test:
   - Should load in < 2 seconds
   - All KPI cards should show real data
   - Real-time updates should work (test with new quote creation)

2. Mobile Responsiveness Test:
   - Test on iPhone SE (smallest screen)
   - All buttons should be tappable
   - No horizontal scroll

3. Bilingual Test:
   - Toggle language to Arabic
   - Layout should flip to RTL
   - Numbers should format correctly (Arabic-Indic numerals)

4. Kanban Board Test:
   - Drag inquiry from "Sent" to "Quoted"
   - Should update database immediately

- Should sync across multiple browser tabs (real-time)

CODE QUALITY REQUIREMENTS:
- Use TypeScript for all components (strict mode)
- Implement error boundaries for graceful error handling
- Use React Query for API state management
- Implement skeleton loading states (no blank screens)
- Add accessibility: ARIA labels, keyboard navigation
- Use Lighthouse to achieve 90+ performance score

IMPORTANT NOTES:
- PocketBase runs as separate process (port 8090)
- Frontend runs on port 3000 (Next.js dev server)
- Use environment variables for API URLs
- Implement token refresh logic (PocketBase tokens expire after 14 days)
- Add CSP headers for security
- Implement rate limiting on frontend (prevent API abuse)

Generate all files with complete, production-ready code. Include setup instructions in README.
```

---

## 🚀 PHASE 6: Deployment, Testing & Production Optimization
PROJECT CONTEXT:
You are implementing Phase 6 of alsakr online: Final production deployment, comprehensive testing, monitoring, and optimization for a self-hosted zero-cost infrastructure.
PREVIOUS PHASES:
✅ Phase 1: Backend infrastructure
✅ Phase 2: Web scraping system
✅ Phase 3: AI multimodal search
✅ Phase 4: RFQ automation
✅ Phase 5: CRM & user dashboard
TECHNICAL REQUIREMENTS:

Deployment: Docker + Docker Compose
Reverse Proxy: Caddy (auto-SSL with Let's Encrypt)
Monitoring: Prometheus + Grafana (self-hosted)
Logging: Loki + Promtail
Backup: Automated database backups to cloud storage

YOUR TASK:
Create the complete production deployment system:

NEW DIRECTORY STRUCTURE:
alsakr-online/
├── docker/
```

```
│   ├── Dockerfile.backend (FastAPI app)
│   ├── Dockerfile.frontend (Next.js app)
│   ├── Dockerfile.ollama (AI models)
│   └── Dockerfile.qdrant (Vector DB)
├── docker-compose.yml (orchestrate all services)
├── docker-compose.prod.yml (production overrides)
├── caddy/
│   └── Caddyfile (reverse proxy config)
├── monitoring/
│   ├── prometheus.yml
│   ├── grafana/
│   │   └── dashboards/ (JSON dashboard configs)
│   └── loki-config.yml
├── scripts/
│   ├── deploy.sh (one-command deployment)
│   ├── backup.sh (database backup script)
│   ├── restore.sh (restore from backup)
│   └── health_check.sh (system health monitor)
├── tests/
│   ├── unit/ (pytest tests)
│   ├── integration/ (API tests)
│   ├── e2e/ (Playwright browser tests)
│   └── load/ (Locust load testing)
└── docs/
    ├── DEPLOYMENT.md
    ├── API.md
    └── TROUBLESHOOTING.md
```

DOCKER COMPOSE CONFIGURATION:
Create docker-compose.yml:

```
yamlversion: '3.8'

services:
  # Backend API
  backend:
    build:
      context: ./backend
      dockerfile: ../docker/Dockerfile.backend
    ports:
      - "8000:8000"
    environment:
      - DATABASE_URL=sqlite:///./data/alsakr.db
      - OLLAMA_HOST=http://ollama:11434
      - QDRANT_HOST=http://qdrant:6333
    volumes:
      - ./backend/data:/app/data
      - ./backend/logs:/app/logs
    depends_on:
```

```
    - ollama
    - qdrant
  restart: unless-stopped

# Frontend
frontend:
  build:
    context: ./frontend
    dockerfile: ../docker/Dockerfile.frontend
  ports:
    - "3000:3000"
  environment:
    - NEXT_PUBLIC_API_URL=http://backend:8000
    - NEXT_PUBLIC_POCKETBASE_URL=http://pocketbase:8090
  depends_on:
    - backend
  restart: unless-stopped

# AI LLM Server
ollama:
  image: ollama/ollama:latest
  ports:
    - "11434:11434"
  volumes:
    - ./data/ollama:/root/.ollama
  deploy:
    resources:
      reservations:
        devices:
          - driver: nvidia
            count: 1
            capabilities: [gpu]
  restart: unless-stopped

# Vector Database
qdrant:
  image: qdrant/qdrant:latest
  ports:
    - "6333:6333"
  volumes:
    - ./data/qdrant:/qdrant/storage
  restart: unless-stopped

# CRM Backend
pocketbase:
  image: ghcr.io/muchobien/pocketbase:latest
  ports:
    - "8090:8090"
```

```
  volumes:
    - ./backend/pocketbase/pb_data:/pb_data
  restart: unless-stopped

# Workflow Automation
n8n:
  image: n8nio/n8n:latest
  ports:
    - "5678:5678"
  environment:
    - N8N_BASIC_AUTH_ACTIVE=true
    - N8N_BASIC_AUTH_USER=admin
    - N8N_BASIC_AUTH_PASSWORD=${N8N_PASSWORD}
  volumes:
    - ./data/n8n:/home/node/.n8n
  restart: unless-stopped

# Reverse Proxy
caddy:
  image: caddy:latest
  ports:
    - "80:80"
    - "443:443"
  volumes:
    - ./caddy/Caddyfile:/etc/caddy/Caddyfile
    - ./data/caddy:/data
  restart: unless-stopped

# Monitoring: Prometheus
prometheus:
  image: prom/prometheus:latest
  ports:
    - "9090:9090"
  volumes:
    - ./monitoring/prometheus.yml:/etc/prometheus/prometheus.yml
    - ./data/prometheus:/prometheus
  command:
    - '--config.file=/etc/prometheus/prometheus.yml'
  restart: unless-stopped

# Monitoring: Grafana
grafana:
  image: grafana/grafana:latest
  ports:
    - "3001:3000"
  environment:
    - GF_SECURITY_ADMIN_PASSWORD=${GRAFANA_PASSWORD}
  volumes:
```

```
      - ./monitoring/grafana:/etc/grafana/provisioning
      - ./data/grafana:/var/lib/grafana
    restart: unless-stopped

  # Logging: Loki
  loki:
    image: grafana/loki:latest
    ports:
      - "3100:3100"
    volumes:
      - ./monitoring/loki-config.yml:/etc/loki/local-config.yaml
      - ./data/loki:/loki
    restart: unless-stopped

  # Logging: Promtail
  promtail:
    image: grafana/promtail:latest
    volumes:
      - ./backend/logs:/var/log
      - ./monitoring/promtail-config.yml:/etc/promtail/config.yml
    command: -config.file=/etc/promtail/config.yml
    restart: unless-stopped

volumes:
  postgres_data:
  ollama_models:
  qdrant_data:

DOCKERFILES:
Create optimized Dockerfiles:

Dockerfile.backend:
dockerfileFROM python:3.11-slim

WORKDIR /app

# Install system dependencies
RUN apt-get update && apt-get install -y \
    gcc g++ libgomp1 \
    && rm -rf /var/lib/apt/lists/*

# Copy requirements
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# Copy application
COPY . .
```

```
# Run migrations and start server
CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "8000"]
Dockerfile.frontend:
dockerfileFROM node:20-alpine AS builder

WORKDIR /app
COPY package*.json ./
RUN npm ci
COPY . .
RUN npm run build

FROM node:20-alpine
WORKDIR /app
COPY --from=builder /app/.next ./.next
COPY --from=builder /app/node_modules ./node_modules
COPY --from=builder /app/package.json ./package.json
COPY --from=builder /app/public ./public

EXPOSE 3000
CMD ["npm", "start"]
```

4. CADDY REVERSE PROXY (Caddyfile):
```
alsakronline.com {
    # Frontend
    reverse_proxy frontend:3000

    # Backend API
    handle /api/* {
        reverse_proxy backend:8000
    }

    # PocketBase CRM
    handle /crm/* {
        reverse_proxy pocketbase:8090
    }

    # n8n Workflows
    handle /workflows/* {
        reverse_proxy n8n:5678
    }

    # Monitoring (admin only)
    handle /grafana/* {
        reverse_proxy grafana:3000
    }
```

```
    # Auto SSL
    tls {
        dns cloudflare {env.CLOUDFLARE_API_TOKEN}
    }

    # Security headers
    header {
        Strict-Transport-Security "max-age=31536000;"
        X-Content-Type-Options "nosniff"
        X-Frame-Options "DENY"
        X-XSS-Protection "1; mode=block"
    }

    # Gzip compression
    encode gzip

    # Logging
    log {
        output file /var/log/caddy/access.log
    }
}
```

MONITORING SETUP:
Create prometheus.yml:

```
yamlglobal:
  scrape_interval: 15s

scrape_configs:
  - job_name: 'backend'
    static_configs:
      - targets: ['backend:8000']
    metrics_path: '/metrics'

  - job_name: 'qdrant'
    static_configs:
      - targets: ['qdrant:6333']

  - job_name: 'caddy'
    static_configs:
      - targets: ['caddy:2019']

  - job_name: 'node-exporter'
    static_configs:
      - targets: ['node-exporter:9100']
```

Create Grafana dashboard (monitoring/grafana/dashboards/alsakr.json):

Panels:

API Request Rate (requests/second)
Response Time (p50, p95, p99 latency)
Error Rate (4xx, 5xx responses)
AI Model Inference Time (milliseconds)
Database Query Time
Scraper Jobs (active/failed)
User Activity (active users, searches per hour)

AUTOMATED TESTING:
Create tests/integration/test_api.py (pytest):

pythonimport pytest
from fastapi.testclient import TestClient
from app.main import app

client = TestClient(app)

def test_health_check():
    response = client.get("/api/health")
    assert response.status_code == 200
    assert response.json()["status"] == "healthy"

def test_text_search():
    response = client.post("/api/search/text", json={
        "query": "motor bearings",
        "language": "en"
    })
    assert response.status_code == 200
    assert len(response.json()["results"]) > 0

def test_image_search():
    with open("tests/fixtures/test_part.jpg", "rb") as f:
        response = client.post("/api/search/image", files={"image": f})
    assert response.status_code == 200
    assert "results" in response.json()

def test_rfq_creation():
    response = client.post("/api/rfq/create", json={
        "part_id": "test-part-123",
        "quantity": 10,
        "urgency": "normal"
    }, headers={"Authorization": "Bearer test-token"})
    assert response.status_code == 201
    assert response.json()["status"] == "sent"

Create tests/e2e/test_user_flow.py (Playwright):
pythonfrom playwright.sync_api import Page

```
def test_full_search_to_rfq_flow(page: Page):
    # 1. Login
    page.goto("https://alsakronline.com/login")
    page.fill("input[name=email]", "test@example.com")
    page.fill("input[name=password]", "password123")
    page.click("button[type=submit]")

    # 2. Search for part
    page.goto("https://alsakronline.com/dashboard")
    page.fill("input[placeholder='Search for parts...']", "ABB circuit breaker")
    page.click("button:has-text('Search')")

    # 3. Select part
    page.click(".part-card:first-child")

    # 4. Request quote
    page.click("button:has-text('Request Quote')")
    page.fill("input[name=quantity]", "5")
    page.click("button:has-text('Send RFQ')")

    # 5. Verify success
    assert page.locator("text=RFQ sent successfully").is_visible()
```

Create tests/load/locustfile.py (load testing):
pythonfrom locust import HttpUser, task, between

```
class alsakrUser(HttpUser):
    wait_time = between(1, 3)

    @task(3)
    def search_parts(self):
        self.client.post("/api/search/text", json={
            "query": "motor bearings",
            "language": "en"
        })

    @task(1)
    def view_dashboard(self):
        self.client.get("/api/dashboard", headers={
            "Authorization": f"Bearer {self.token}"
        })

    def on_start(self):
        # Login and get token
        response = self.client.post("/api/auth/login", json={
            "email": "test@example.com",
```

```
            "password": "password123"
        })
        self.token = response.json()["access_token"]
```

DEPLOYMENT SCRIPT (scripts/deploy.sh):

```
bash#!/bin/bash
set -e

echo "🚀 Deploying alsakr online..."

# 1. Pull latest code
git pull origin main

# 2. Build Docker images
docker-compose -f docker-compose.yml -f docker-compose.prod.yml build

# 3. Stop old containers
docker-compose down

# 4. Start new containers
docker-compose -f docker-compose.yml -f docker-compose.prod.yml up -d

# 5. Run database migrations
docker-compose exec backend python -m alembic upgrade head

# 6. Download AI models (if not cached)
docker-compose exec ollama ollama pull llama3.2-vision
docker-compose exec ollama ollama pull llama3.2

# 7. Initialize vector database (if empty)
docker-compose exec backend python scripts/initialize_ai.py

# 8. Health check
sleep 10
curl -f http://localhost:8000/api/health || exit 1

echo "✅ Deployment successful!"
echo "📊 Dashboard: https://alsakronline.com"
echo "📈 Monitoring: https://alsakronline.com/grafana"
```

BACKUP SCRIPT (scripts/backup.sh):

```
bash#!/bin/bash
BACKUP_DIR="/backups/alsakr-$(date +%Y%m%d-%H%M%S)"
mkdir -p $BACKUP_DIR

# Backup SQLite databases
```

cp backend/data/alsakr.db $BACKUP_DIR/
cp backend/pocketbase/pb_data/data.db $BACKUP_DIR/

Backup Qdrant vector DB
docker-compose exec qdrant curl -X POST
http://localhost:6333/collections/parts_text/snapshots
docker cp qdrant:/qdrant/storage/snapshots $BACKUP_DIR/

Backup uploaded files
tar -czf $BACKUP_DIR/uploads.tar.gz backend/data/images backend/data/audio

Upload to cloud (optional)
aws s3 sync $BACKUP_DIR s3://alsakr-backups/

echo "✅ Backup completed: $BACKUP_DIR"

HEALTH CHECK SCRIPT (scripts/health_check.sh):

bash#!/bin/bash

check_service() {
    if curl -sf $1 > /dev/null; then
        echo "✅ $2 is healthy"
    else
        echo "❌ $2 is down!"
        return 1
    fi
}

check_service "http://localhost:8000/api/health" "Backend API"
check_service "http://localhost:3000" "Frontend"
check_service "http://localhost:6333/readyz" "Qdrant"
check_service "http://Continue9:43 PMlocalhost:11434" "Ollama"
check_service "http://localhost:8090/api/health" "PocketBase"

10. ENVIRONMENT VARIABLES (.env.production):
```bash
# Backend
DATABASE_URL=sqlite:///./data/alsakr.db
JWT_SECRET=your-secret-key-change-in-production
OLLAMA_HOST=http://ollama:11434
QDRANT_HOST=http://qdrant:6333

# Email (RFQ automation)
SMTP_SERVER=smtp.gmail.com
SMTP_PORT=587
EMAIL_USERNAME=your-email@gmail.com
EMAIL_PASSWORD=your-app-password
```

```
# Cloudflare (for SSL)
CLOUDFLARE_API_TOKEN=your-cloudflare-token

# n8n
N8N_PASSWORD=secure-admin-password

# Grafana
GRAFANA_PASSWORD=secure-grafana-password

# Frontend
NEXT_PUBLIC_API_URL=https://alsakronline.com/api
NEXT_PUBLIC_POCKETBASE_URL=https://alsakronline.com/crm
```

11. PERFORMANCE OPTIMIZATION:
Add to backend (app/middleware.py):
- Response caching (Redis or in-memory for frequent searches)
- Request compression (gzip)
- Rate limiting (SlowAPI: 100 requests/minute per IP)
- Database connection pooling
- Lazy loading for AI models (load on first use)

Add to frontend:
- Image optimization (Next.js Image component)
- Code splitting (dynamic imports)
- Service Worker for offline support
- CDN for static assets (Cloudflare Pages)

12. SECURITY HARDENING:
- Enable HTTPS only (redirect HTTP → HTTPS)
- Implement CSRF protection (FastAPI CSRF middleware)
- SQL injection prevention (use parameterized queries)
- XSS protection (sanitize user inputs)
- Implement API key rotation
- Add IP whitelisting for admin endpoints
- Enable 2FA for admin accounts (TOTP)

13. DOCUMENTATION (docs/DEPLOYMENT.md):
Include:
- Server requirements (min 4GB RAM, 50GB storage, GPU optional)
- Step-by-step installation guide
- Troubleshooting common issues
- Backup and restore procedures
- Scaling guide (horizontal scaling with load balancer)
- Update/upgrade procedures

DELIVERABLES:

✅ Complete Docker Compose setup (all services)
✅ Automated deployment script (one-command deploy)
✅ Monitoring dashboards (Prometheus + Grafana)
✅ Comprehensive test suite (unit, integration, e2e, load)
✅ Automated backup system
✅ Production-ready security configuration
✅ Complete documentation

VALIDATION TESTS:
1. Full Deployment Test:
   - Run deploy.sh on fresh server
   - Verify all 11 services start successfully
   - Access frontend at https://alsakronline.com
   - Verify SSL certificate is valid

2. Load Test:
   - Run: `locust -f tests/load/locustfile.py --host=https://alsakronline.com`
   - Target: 100 concurrent users
   - Verify: <2s average response time, <1% error rate

3. Backup/Restore Test:
   - Create test data
   - Run backup.sh
   - Delete database
   - Run restore.sh
   - Verify data integrity

4. Monitoring Test:
   - Trigger artificial error (stop Qdrant service)
   - Verify alert appears in Grafana
   - Check Prometheus scrapes metrics correctly

CODE QUALITY REQUIREMENTS:
- All scripts should have error handling (set -e)
- Add logging to all deployment scripts
- Implement rollback mechanism (deploy.sh --rollback)
- Docker images should use multi-stage builds (reduce size)
- Add health checks to all Docker services
- Implement graceful shutdown (handle SIGTERM)

IMPORTANT NOTES:
- Test deployment on staging environment first
- Always backup before deploying to production
- Monitor logs during first 24 hours after deployment
- Implement blue-green deployment for zero-downtime updates
- Keep documentation updated with every change
- Set up automated weekly backups (cron job)

Generate all files with complete, production-ready code. Include troubleshooting guide.

🎯 FINAL SUMMARY: Complete Implementation Roadmap
Estimated Timeline:

Phase 1: 1 week (Foundation)
Phase 2: 2 weeks (Web Scraping)
Phase 3: 2 weeks (AI Search)
Phase 4: 1.5 weeks (RFQ Automation)
Phase 5: 2 weeks (CRM & Dashboard)
Phase 6: 1 week (Deployment & Testing)

Total: ~9.5 weeks for MVP
Success Metrics:

✅ 10,000+ parts scraped from 5 manufacturers
✅ <2s search response time
✅ 95%+ image identification accuracy
✅ 100+ active users within 3 months
✅ 99.9% uptime

Next Steps After MVP:

Mobile app (React Native)
WhatsApp Business integration
Predictive maintenance AI
Multi-currency payment processing
Enterprise ERP integrations

This complete blueprint is ready for implementation in AnythingLLM IDE!