Objective:
My goal is to implement the complete, functional Python backend for the "Price Intelligence (P.I.)" application. The project structure and Docker environment are already defined. You are to generate the Python code to fill in the empty backend files.

Core Requirements & Sources of Truth:
All generated code must be a direct implementation of the features and logic described in these three documents:

1. `Requirements - P.I. - Price Intelligence.pdf` (Functional logic)
2. `UIUX - P.I. - Price Intelligence.pdf` (User stories and technical goals)
3. `Product Analysis AI System Overview (Streamlined with Full Prompts).pdf` (The definitive AI logic, prompts, and JSON schemas)

Technical Architecture:

- **Framework:** FastAPI
- **AI Orchestration:** CrewAI & LangChain
- **Web Server:** Uvicorn
- **Containerization:** Docker & Docker Compose
- **Database/Auth:** Supabase
- **Payments:** Stripe
- **Caching:** Redis

Task: Generate Python Code
Please generate the complete, production-quality Python code for the following files within the `api/src` directory. The code must be fully functional, including all necessary imports, Pydantic models, error handling, and documentation strings.

File-by-File Implementation Plan:

1. Configuration (`config.py` and `dependencies.py`)

- **`api/src/config.py`** : Create a `Settings` class using `pydantic-settings` to load all environment variables from the `.env` file. This must include keys for `GOOGLE_VISION_API_KEY` , `OPENAI_API_KEY` , `STRIPE_SECRET_KEY` , `STRIPE_WEBHOOK_SECRET` , `SUPABASE_URL` , `SUPABASE_SERVICE_KEY` , and `REDIS_URL` .
- **`api/src/dependencies.py`** : Implement FastAPI dependency injection functions to provide instances of the settings, the Supabase client, and the Redis client to the rest of the application.

2. Services Layer (The "Doers")

- **`api/src/services/vision_service.py`** : Implement a `VisionService` class. It must have an `async def analyze_image(base64_image: str)` method that uses the `google-cloud-vision` library

to perform `LABEL_DETECTION`, `TEXT_DETECTION`, and `OBJECT_LOCALIZATION`. It should return a structured dictionary of the results.

- **`api/src/services/marketplace_service.py`**: Implement a `MarketplaceService` class. For now, create a placeholder method `async def fetch_market_data(query: str)` that returns realistic mock data for product prices (e.g., a list of dictionaries with `price` and `source` keys).
- **`api/src/services/fee_service.py`**: Implement a `FeeService` class with a method `get_platform_fees()`. This method must return the fee data for all platforms as detailed in the "Unified Platform Fee Table" from `Product Analysis AI System Overview.pdf`.
- **`api/src/services/cache_service.py`**: Implement the `CacheService` using the Redis client. It needs two methods: `set_analysis(task_id: str, data: dict)` and `get_analysis(task_id: str) -> dict | None`.

3. Tools Layer (LangChain Bridge)

- **`api/src/tools/*.py`**: Create LangChain `Tool` definitions that wrap the methods from the Services layer. For instance, in `vision_tools.py`, create a `vision_analysis_tool` that calls the `VisionService.analyze_image` method. This makes your services usable by AI agents.

4. AI Orchestration Layer (The "Thinkers")

- **`api/src/agents/*.py`**:
  - Define the CrewAI `Agent`s for `VisionAnalyst`, `MarketResearchAnalyst`, and `PlatformRecommendationAnalyst`.
  - Each agent must have a clear `role`, `goal`, `backstory`, and be assigned the appropriate `Tool`s created in the previous step. The backstory for the recommendation agent must emphasize its expertise in maximizing seller profit by analyzing platform fees.
- **`api/src/agents/crew.py`**:
  - Define the main `ProductAnalysisCrew`.
  - Define the sequence of `Tasks` to be executed:
    1. The `VisionAnalyst` analyzes the image to identify the product.
    2. The `MarketResearchAnalyst` uses the product identity to find price ranges.
    3. The `PlatformRecommendationAnalyst` uses the pricing and product type to recommend the best platform, calculating net profit after fees.
    4. A final agent/task must take all the structured data and use an LLM (via a `ContentGenerationTool`) to generate the final human-readable content: `detailedDescription`, `generatedTitle`, and `tagsKeywords`.
  - The final output from the crew's `kickoff` method **must** be a single JSON object that strictly conforms to the `Output Schema` defined for the `analyzeProductImage` function in `Product Analysis AI System Overview.pdf`.

5. API Layer (The "Interface")

- **`api/src/models/*.py`** : Create Pydantic models (`AnalysisResult` , `ProductDetails`) that match the final JSON output schema. These are for data validation and API documentation.
- **`api/src/routers/analysis.py`** :
  - Implement a `POST /analyze` endpoint. It must accept a file upload (`image`) and an optional `condition` string.
  - It must run the `ProductAnalysisCrew` in a **FastAPI `BackgroundTask`** .
  - It should immediately return a JSON response with a unique `task_id` and a `status: "processing"` .
  - Implement a `GET /analyze/result/{task_id}` endpoint that the frontend can poll. This endpoint will check the Redis cache using the `task_id` and return the final analysis once it's available, or a "processing" status if not.
- **`api/src/routers/subscription.py`** :
  - Implement a `POST /stripe-webhook` endpoint. It must securely verify the webhook signature using `STRIPE_WEBHOOK_SECRET` .
  - It needs to handle `invoice.paid` and `customer.subscription.deleted` events to update the user's `credits` or subscription status in the Supabase `users` and `subscriptions` tables.
- **`api/src/main.py`** :
  - Update the main FastAPI app instance to include all the routers (`analysis` , `subscription` , etc.).
  - Configure CORS to allow requests from the frontend service.
  - Add a root `/` endpoint that returns a simple "Welcome" message for health checks.