Project Outline: Gemini Trader

An autonomous trading agent using LangGraph, Gemini, and the Bybit API to execute strategies on the 15-minute timeframe.

Phase 1: Foundation & Environment Setup

Goal: Establish a secure and robust foundation for interacting with the exchange and processing market data. All exchange-related logic should be modular and separate from the AI logic.

- **1. Project Structure & Environment**
  - Create a root project folder (e.g., gemini-trader).
  - Initialize a Python virtual environment: python -m venv venv and activate it.
  - Create primary Python files: main.py, bybit_tools.py, data_processor.py, agent_tools.py, prompts.py, graph.py.
  - Install core libraries:
    pip install langchain langchain-google-genai langgraph pybit pandas pandas-ta python-dotenv schedule

  - Create a .env file for API keys. **Never commit this file to Git.**
    BYBIT_API_KEY_TESTNET="YOUR_TESTNET_KEY"
    BYBIT_API_SECRET_TESTNET="YOUR_TESTNET_SECRET"
    GOOGLE_API_KEY="YOUR_GOOGLE_AI_STUDIO_KEY"

- **2. Bybit Connector (bybit_tools.py)**
  - A dedicated module for all Bybit API interactions.
  - **Crucially, initialize the Bybit session using a DEMO/TESTNET account to prevent financial risk during development and testing.** The pybit library makes this easy:

```
# Example of testnet session initialization in bybit_tools.py
import os
from pybit.unified_trading import HTTP

session = HTTP(
    testnet=True,
    api_key=os.environ.get("BYBIT_API_KEY_TESTNET"),
    api_secret=os.environ.get("BYBIT_API_SECRET_TESTNET"),
)
```

  - **Functions to implement:**

        - get_market_data(symbol, interval, limit): Fetches historical OHLCV data.
        - get_account_balance(account_type): Retrieves the current balance to calculate position sizes.
        - place_market_order(symbol, side, qty): Executes a trade.
        - get_open_positions(symbol): Checks if a position is already open to avoid duplicate trades.
        - close_position(symbol): Logic to place an opposing order to exit a trade.
- **3. Data Processor (data_processor.py)**
    - A module to enrich the raw data from Bybit.
    - **Function to implement:**
        - add_technical_indicators(dataframe):
            - Accepts a pandas DataFrame of OHLCV data.
            - Uses the pandas-ta library to append columns for RSI, MACD (macd, macdh, macds), and Bollinger Bands (bbl, bbm, bbu).
            - Returns the enriched DataFrame.

Phase 2: The Agent's "Brain" - AI & Strategy

Goal: Define the agent's capabilities (tools) and its core trading logic (the prompt).

- **1. Agent Tool Definition (agent_tools.py)**
    - This module exposes your foundational functions to the LangChain agent.
    - Import functions from bybit_tools.py and data_processor.py.
    - Use LangChain's @tool decorator.
    - **Tool to create:**
        - analyze_market_state(symbol: str, interval: int) -> str:
            - This is the primary tool the agent will call.
            - It orchestrates the process: calls get_market_data, passes the result to add_technical_indicators.
            - Formats the last few rows of the enriched DataFrame into a clean, LLM-readable string or JSON object.
            - Crucially, it should also call get_open_positions to inform the LLM if it is already in a trade.
- **2. Prompt Engineering (prompts.py)**
    - This is the heart of your agent's logic.
    - Create a system_prompt variable containing a detailed multi-part prompt.
    - **Key Sections of the Prompt:**
        - **Persona:** "You are 'Gemini Trader,' an analytical and cautious automated trading agent. You operate exclusively on the 15-minute chart."
        - **Core Strategy:** Define the exact conditions for entering a trade. Be explicit. (e.g., "A BUY signal is only valid if the RSI is below 35, the MACD histogram has just crossed above zero, and the price is touching the lower Bollinger Band.")
        - **Risk Management:** "Always calculate position size to risk no more than 2% of the total account balance. The stop loss is 0.75% below the entry price."

        - **State Awareness:** "You must check if a position is already open for the given symbol. If a position is open, your only valid actions are 'HOLD' or 'CLOSE'."
        - **Output Formatting:** "Your final response MUST be a JSON object. Do not add any other text. The format is: {\"action\": \"BUY|SELL|HOLD|CLOSE\", \"quantity\": float, \"reasoning\": \"Your detailed analysis here.\"}"

Phase 3: Orchestration with LangGraph

Goal: Build a state machine that controls the agent's workflow, ensuring a predictable and logical sequence of operations.

- **1. Graph State Definition (graph.py)**
    - Define a TypedDict for the graph's state.
    - **State Fields:** symbol: str, interval: int, market_analysis: str, llm_decision: dict, trade_executed: bool, error_message: str.
- **2. Graph Node Implementation (graph.py)**
    - Implement functions for each step in the workflow.
    - **Nodes:**
        - analyze_market(state): Calls the analyze_market_state tool and populates the market_analysis field in the state.
        - make_trade_decision(state): Constructs the prompt using the system_prompt and the market_analysis. Calls the Gemini model. Parses the JSON response and populates llm_decision.
        - execute_trade(state): Checks the action from llm_decision. If it's BUY or SELL, it calls the place_market_order tool and sets trade_executed to True.
- **3. Graph Construction (graph.py)**
    - Instantiate StatefulGraph.
    - Add the nodes: analyze_market, make_trade_decision, execute_trade.
    - Define the edges to control the flow:
        - Set analyze_market as the entry point.
        - Edge: analyze_market -> make_trade_decision.
        - **Conditional Edge** from make_trade_decision:
            - If action is BUY or SELL -> go to execute_trade.
            - If action is HOLD or CLOSE -> go to END.
        - Edge: execute_trade -> END.
    - Compile the graph using .compile().

Phase 4: Execution & Monitoring

Goal: Run the agent on a schedule and maintain detailed logs for debugging and performance review.

- **1. Main Execution Loop (main.py)**
    - Import the compiled graph from graph.py.
    - Load environment variables using dotenv.
    - Define the main trading function run_trading_cycle().

        - Inside, invoke the graph with the initial state (e.g., {"symbol": "BTCUSDT", "interval": 15}).
        - Print or log the final state of the graph.
    - Use the schedule library to run run_trading_cycle every 15 minutes.
    - Create a while True loop to keep the scheduler running.
- **2. Logging and Monitoring**
    - Integrate Python's built-in logging module.
    - Configure it to write to both the console and a file (e.g., trading_log.log).
    - **What to log:**
        - Start and end of each trading cycle.
        - The market analysis sent to the LLM.
        - The raw JSON decision from the LLM.
        - Confirmation of trade execution from Bybit.
        - Any errors that occur in any step.