IMPLEMENTED DESIGN PATTERNS

Command Pattern

When developing software that supports multiple, separate analyses — such as fundamental sentiment analysis, technical indicator-based analysis, and LSTM machine-learning predictions — a common challenge is ensuring that each type of analysis can be run independently.

```
class AnalysisCommand(ABC):
    """
    The Command interface, declaring the execute() method.
    Each analysis (fundamental, technical, LSTM) will implement this.
    """

    @abstractmethod
    def execute(self):
        pass
```

The Command pattern proves especially powerful. The fundamental idea of Command is to encapsulate each request — each discrete action you want to perform — into its own "Command object." We do this by defining an interface, typically called AnalysisCommand, that specifies a single method: execute(). Each unique analysis then becomes a Concrete Command that implements this interface. In other words, for fundamental sentiment analysis, we have a class FundamentalAnalysisCommand whose execute() method calls our existing function for sentiment-based signals. For technical analysis, we have TechnicalAnalysisCommand, which runs our existing perform_technical_analysis() code. And for machine-learning predictions, we have LSTMCommand, which calls the relevant LSTM training and prediction logic.

```
class FundamentalAnalysisCommand(AnalysisCommand):
    def __init__(self, company_code):
        self.company_code = company_code

    def execute(self):
        result = get_fundamental_analysis(self.company_code)
        return result
```

```
class TechnicalAnalysisCommand(AnalysisCommand):
    def __init__(self, db_conn, issuer, timeperiod=30):
        self.db_conn = db_conn
        self.issuer = issuer
        self.timeperiod = timeperiod

    def execute(self):
        result = perform_technical_analysis(self.db_conn, self.issuer,
self.timeperiod)
        return result
```

```python
class LSTMCommand(AnalysisCommand):
    def __init__(self, issuer, train_ratio=0.7, rolling_window=5,
                 n_units=50, dropout_rate=0.2, epochs=20, batch_size=32):
        self.issuer = issuer
        self.train_ratio = train_ratio
        self.rolling_window = rolling_window
        self.n_units = n_units
        self.dropout_rate = dropout_rate
        self.epochs = epochs
        self.batch_size = batch_size

    def execute(self):
        prediction, chart_base64 = train_and_evaluate_stock_model_with_image(
            stock_symbol=self.issuer,
            train_ratio=self.train_ratio,
            rolling_window=self.rolling_window,
            n_units=self.n_units,
            dropout_rate=self.dropout_rate,
            epochs=self.epochs,
            batch_size=self.batch_size
        )
        return {"prediction": prediction, "image": chart_base64, "error":
None}
```

By doing so, each analysis exists as its own "capsule of behavior," holding all the parameters and processes needed to execute that analysis. We then introduce an Invoker class—something like AnalysisInvoker—that has a straightforward method execute_command(command) which simply invokes command.execute(). In practice, this means our client code (for example, our Flask routes) can create whichever command object it needs based on the user's request, pass it to the invoker, and then yield the result.

```python
class AnalysisInvoker:
    """
    The Invoker in the Command pattern.
    This class is responsible for invoking commands that follow the Command
pattern.
    """

    def execute_command(self, command: AnalysisCommand):
        return command.execute()
```

Example command invocation:

```python
cmd = TechnicalAnalysisCommand(conn, issuer, timeperiod)
result = analysis_invoker.execute_command(cmd)
```

Implementing the Command pattern yields multiple benefits. First, it increases flexibility: each route or each client call can choose precisely which analysis to run. Second, it improves maintainability: if the logic of our fundamental analysis changes, we only modify the FundamentalAnalysisCommand class, without touching technical or LSTM code. Third, it scales: if we want to add another analysis type tomorrow—say, a purely statistical method to forecast stock prices—we can introduce a new command (e.g., StatisticalAnalysisCommand) and integrate it seamlessly. Fourth, it reduces coupling: the Flask application or any other "client" that triggers analyses doesn't need to know the detailed steps or parameters inside each analysis, only how to construct the relevant command. Finally, potential expansions like scheduling these commands for later, retrying them, or even implementing undo/redo can all flow quite naturally from the Command pattern's structure.

Singleton Pattern

In our application, we have also implemented the Singleton Pattern, to manage database connections efficiently and consistently. The DatabaseConnection class ensures that only one instance of the database connection manager exists across the entire application. This approach centralizes connection logic, reduces redundancy, and simplifies maintenance while addressing SQLite's threading limitations.
To handle SQLite's restriction against sharing connections across threads, we use threading.local. This allows each thread to have its own isolated connection while maintaining the Singleton instance. The get_connection() method creates a thread-local connection if one does not already exist, ensuring proper isolation and avoiding conflicts. The close_connection() method ensures that each thread's connection is closed and cleaned up when no longer needed.

```
class DatabaseConnection:
    _instance = None
    _lock = Lock()
    _thread_local = local()

    def __new__(cls):

        if cls._instance is None:
            with cls._lock:
                if cls._instance is None:
                    cls._instance = super(DatabaseConnection,
cls).__new__(cls)
                    cls._instance._conn = None
        return cls._instance

    def get_connection(self):
        if not hasattr(self._thread_local, "connection"):
            try:
                self._thread_local.connection = sqlite3.connect(DB_PATH,
check_same_thread=False)
                self._thread_local.connection.row_factory = sqlite3.Row
                print("Database connection established successfully.")
            except sqlite3.Error as e:
                print(f"Error connecting to the database: {e}")
                self._thread_local.connection = None
        return self._thread_local.connection

    def close_connection(self):
        if hasattr(self._thread_local, "connection") and
self._thread_local.connection:
            self._thread_local.connection.close()
            self._thread_local.connection = None
            print("Database connection closed.")
```

Usage:

```
db_instance = DatabaseConnection()
connection = db_instance.get_connection()
```

This design supports scalability, allowing seamless integration of future features like connection pooling or monitoring while maintaining consistent and reliable database access across the application.