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.

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 LSTMAnalysisCommand, which calls the relevant LSTM training and prediction logic.

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.

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.