1. User Interface (UI):
   * Determine the type of UI you want to create (WEB BASED as it will run on the cloud and this is a standard setup).
   * Design an intuitive and user-friendly interface that allows users to input the necessary parameters and view the bot's output (Maybe figma or some tool to sketch something and convert to code?)
   * Consider using a framework or library to streamline the UI development process (e.g., Flask for web-based UI, PyQt for desktop applications). (Not sure about this part yet)
2. User Input Parameters:
   * Trading Pair: Allow users to specify the trading pair they want to trade (e.g., BTC/USDT, ETH/USDC).
   * Entry Condition: Let users define the condition that triggers the bot to enter a trade. This could include:
     + Price-based conditions (e.g., when the price reaches a certain level)
     + Volume-based conditions (e.g., when the trading volume exceeds a specific threshold)
     + Percentage-based conditions (e.g., when the price moves by a certain percentage)
     + Technical indicators (e.g., when a specific technical indicator crosses a certain level)
   * Exit Condition: Allow users to specify the condition for exiting a trade, similar to the entry condition.
   * Trade Amount: Let users input the amount they want to trade (e.g., in terms of the base currency or a specific quantity).
   * Source and Destination Exchanges: Allow users to select the exchanges they want to trade on.
   * Slippage Tolerance: Let users specify the maximum slippage they are willing to accept for their trades.
   * Gas Price (for DeFi): Allow users to set the gas price they are willing to pay for transactions on the Ethereum network.
3. Replace the Existing Bridge with L1DEX (do this before UI/UX):
   * Research the L1DEX bridge and understand its API and functionality.
   * Modify the func\_bridge.py file to integrate L1DEX instead of the current bridge implementation.
   * Update the relevant functions in func\_entry\_pairs.py and func\_exit\_pairs.py to use the L1DEX bridge for cross-platform arbitrage.
   * Test the integration thoroughly to ensure seamless functionality.
4. Optimize for Speed (Must be async, most already is, I will see how hard it is to incorp multithreading, Websocket, caching):
   * Identify the performance bottlenecks in the current implementation.
   * Optimize the code for speed by:
     + Using asynchronous programming techniques (e.g., asyncio) to handle multiple tasks concurrently.
     + Minimizing network latency by using WebSocket connections instead of REST APIs whenever possible.
     + Caching frequently accessed data to reduce redundant API calls.
     + Optimizing algorithms and data structures to minimize computational overhead.
   * Continuously monitor and profile the bot's performance to identify areas for further optimization.
5. DeFi Considerations (This is important, but first I should get the basics of the bot in place fully):
   * Research the specific DeFi platforms and protocols you plan to integrate with.
   * Understand the unique challenges and considerations for trading in the DeFi space, such as gas costs, network congestion, and flash loan opportunities.
   * Adapt the bot's architecture and strategies to accommodate the fast-paced and competitive nature of DeFi trading.
   * Implement robust error handling and risk management mechanisms to handle the inherent volatility and risks associated with DeFi.