**Detailed Explanation of Key Files**

**Core Configuration Files**

1. **Cargo.toml**
   * This file defines your project's metadata, dependencies, and build configurations
   * Key dependencies will include:
     + tokio for async runtime
     + web3 or ethers-rs for Ethereum/Arbitrum interaction
     + serde for serialization/deserialization
     + tungstenite for WebSocket connections
     + dashmap for concurrent data structures
     + Custom IPC libraries
2. **src/config.rs**
   * Manages all configuration settings for the application
   * Implements loading from environment variables and config files
   * Contains settings for:
     + WebSocket endpoints
     + IPC connection details
     + RPC endpoints
     + Trading pairs to monitor
     + Liquidity tier thresholds
     + Minimum profitability thresholds

**Node Communication Layer**

1. **src/node/ipc.rs**
   * Implements IPC communication with the Arbitrum node
   * Manages shared memory segments for high-performance data exchange
   * Handles connection establishment and maintenance
   * Implements error handling and reconnection logic
2. **src/node/websocket.rs**
   * Manages WebSocket connections to the Arbitrum node
   * Implements subscription to head changes
   * Processes incoming WebSocket messages
   * Handles connection management and reconnection
   * Optimized for low-latency performance
3. **src/node/rpc.rs**
   * Implements RPC communication with the Arbitrum node
   * Handles batch RPC requests for liquidity fetching
   * Manages request rate limiting
   * Implements error handling and retry logic

**Scanner Core**

1. **src/scanner/engine.rs**
   * Core scanning logic that processes WebSocket data
   * Implements parallel processing of 30 trading pairs
   * Coordinates with the database for liquidity prefetching
   * Triggers profitability calculations when potential opportunities are detected
   * Routes valid arbitrage opportunities to the execution module
2. **src/scanner/liquidity\_tiers.rs**
   * Implements the liquidity tier logic as specified in requirements
   * Categorizes pairs based on their liquidity
   * Manages the minimum price difference thresholds for each tier
   * Provides dynamic adjustment of thresholds based on market conditions
3. **src/scanner/profitability.rs**
   * Calculates potential profit for arbitrage opportunities
   * Accounts for:
     + Slippage based on trade size and liquidity
     + Gas fees on Arbitrum
     + Trading fees on each DEX
   * Implements optimized math for fast calculations

**Exchange Integration**

1. **src/exchanges/common.rs**
   * Common interface and shared code for all exchanges
   * Defines traits that each exchange implementation must follow
   * Shared utilities for price and liquidity calculations
2. **src/exchanges/uniswap.rs**, **src/exchanges/sushiswap.rs**, **src/exchanges/camelot.rs**
   * Exchange-specific implementations
   * Contract address management
   * Pool data processing
   * Price calculation logic specific to each exchange
   * Function signature definitions for smart contract interactions

**Execution Interface**

1. **src/execution/ipc\_bridge.rs**
   * Handles IPC communication with the execution bot
   * Implements efficient serialization of arbitrage opportunities
   * Manages shared memory segments for high-performance data exchange
   * Includes acknowledgment and error handling
2. **src/execution/signal.rs**
   * Formats and validates arbitrage signals before sending to execution bot
   * Implements prioritization logic if multiple opportunities are detected
   * Tracks signal status and responses

**Database**

1. **src/database/in\_memory.rs**
   * Fast in-memory storage for liquidity data
   * Implements concurrent access with minimal locking
   * Optimized for frequent reads and periodic writes
   * Uses efficient data structures for quick lookups
2. **src/database/persistent.rs**
   * Optional persistent storage for historical data
   * Useful for analysis and optimization
   * Implements efficient serialization and storage

**Utils**

1. **src/utils/logging.rs**
   * Implements structured logging
   * Configurable log levels
   * Performance optimized for minimal overhead in production
2. **src/utils/metrics.rs**
   * Tracks performance metrics:
     + WebSocket latency
     + Opportunity detection time
     + Signal processing time
   * Provides real-time visibility into bot performance

**Implementation Strategy**

Since your project has several components that need to work together seamlessly, I recommend implementing it in phases:

**Phase 1: Infrastructure**

1. Set up the Rust project structure
2. Implement IPC communication with the Arbitrum node
3. Establish WebSocket connections and verify data flow
4. Implement the in-memory database for liquidity storage

**Phase 2: Core Functionality**

1. Implement basic scanner engine
2. Add exchange-specific price and liquidity calculations
3. Implement liquidity tier and profitability logic
4. Create the IPC bridge to the execution bot

**Phase 3: Optimization and Testing**

1. Optimize WebSocket processing for minimal latency
2. Implement comprehensive test suite
3. Add performance metrics and monitoring
4. Fine-tune thresholds and parameters

**Key Rust-Specific Design Considerations**

1. **Concurrency Model**: Use Tokio's async/await for concurrency, with carefully designed task boundaries to maximize throughput.
2. **Memory Management**: Utilize Rust's ownership model to eliminate data races while minimizing copying. Use Arc/Mutex where shared ownership is needed.
3. **Error Handling**: Implement custom error types with context information for robust error handling and debugging.
4. **Zero-Copy Deserialization**: When possible, use zero-copy deserialization for WebSocket data to minimize latency.
5. **Channels**: Use Tokio's MPSC channels for communication between components, with appropriate buffer sizes to handle bursts of activity.
6. **Shared Memory**: Implement efficient IPC using shared memory with appropriate synchronization primitives.