📊 Binance Futures Trading Bot - Technical Report

Developer: Raman Singh **Date:** January 21, 2026 **Repository:** raman-binance-bot

1. Project Overview

This project implements a production-grade CLI-based trading bot for Binance USDT-M Futures with comprehensive support for both fundamental and advanced order types. Built with safety, modularity, and professional logging as core principles, the system enables automated trading strategies while maintaining strict input validation and error handling protocols.

The bot successfully implements all mandatory core orders (Market and Limit) alongside three advanced strategies: OCO (One-Cancels-the-Other) with WebSocket monitoring, Grid Trading for range-bound markets, and TWAP (Time-Weighted Average Price) for large order execution.

2. Architecture & Technical Design

2.1 Technology Stack

Language
Python 3.14 with type hints and modular design patterns

Package Manager
uv for fast, reliable dependency resolution

API Connector
binance-futures-connector v4.1.0 with WebSocket support

Configuration
python-dotenv for secure environment variable management

Logging System

Dual-channel logging (Console + bot.log file)

Testing

Binance Futures Testnet with real-time validation

2.2 Project Structure

The codebase follows a modular architecture separating core order logic from advanced strategies:

```
cli-trader/ ├── src/ │ ├── market_orders.py # Core: Instant market execution │ ├── limit_orders.py # Core: Price-triggered execution │ ├── advanced/ │ │ ├── oco.py # WebSocket-managed OCO brackets │ │ ├── grid.py # Automated range trading │ │ ├── twap.py # Time-weighted order splitting │ │ └── stop_limit.py # Conditional limit orders │ └── utils/ │ ├── config.py # API client & logger setup │ └── validation.py # Input validation functions ├── .env # API credentials (gitignored) ├── bot.log # Execution logs ├── README.md # Documentation └── requirements.txt # Dependencies
```

2.3 Security Implementation

- **Testnet-First Approach:** All testing conducted on Binance Futures Testnet (USE_TESTNET=True)
- **Environment Variables:** API keys stored in .env file, excluded from version control via .gitignore
- **API Permissions:** Limited to Futures trading only, no withdrawal capabilities
- **Input Validation:** Comprehensive validation for symbols, quantities, and prices before API calls

3. Core Order Implementation & Testing

3.1 Market Orders

Objective: Execute immediate buy/sell orders at current market price for quick

position entry/exit.

> **Command:**
> `uv run python src/market_orders.py BTCUSDT BUY 0.002`

Test Execution: Multiple market buy orders were placed to build an initial position. Orders executed at approximately $90,157 and $90,133 with instant fills.

📸 *Screenshot 1: Terminal showing Market Order execution logs (Image 8)*

📸 *Screenshot 2: Binance Testnet showing filled Market orders at 21:08:40 and 21:11:53 (Image 2)*

Observation: On the Testnet, occasional latency in the matching engine caused some orders to remain in "NEW" status briefly. The logging system captured all state transitions for debugging.

3.2 Limit Orders

Objective: Place orders that execute only at a specified price or better, useful for patient trading and avoiding slippage.

> **Command:**
> `uv run python src/limit_orders.py BTCUSDT BUY 0.005 90500`

Test Results:

- Limit Buy placed at $90,500 (Order ID: 11854144928)
- Successfully filled when market price reached the limit level
- Average fill price: $90,262.90

📸 *Screenshot 3: Terminal log showing Limit Order placement with full API response (Image 1)*

📸 *Screenshot 4: Binance Order History showing filled Limit orders (Image 6)*

4. Advanced Strategies (Bonus Implementation)

4.1 OCO (One-Cancels-the-Other) with WebSocket

Challenge: Unlike Binance Spot, Futures does not support native OCO orders. This required implementing a client-side WebSocket manager.

Implementation Approach:

1. Create a user data stream and obtain listen key from Binance API
2. Place both Take Profit (Limit) and Stop Loss (Stop Market) orders simultaneously
3. Connect to Binance WebSocket to receive real-time ORDER_TRADE_UPDATE events
4. Monitor both orders; when one fills, automatically cancel the sibling order
5. Gracefully handle interruptions and ensure cleanup

> **Command:**
> `PYTHONPATH=src uv run python src/advanced/oco.py BTCUSDT 0.005 89840 89740`

Test Scenario:

- Position Size: 0.005 BTC
- Take Profit: $89,840 (Limit Sell)
- Stop Loss: $89,740 (Stop Market trigger)

📸 *Screenshot 5: Terminal showing OCO bracket setup and WebSocket connection (Image 4)*

📸 *Screenshot 6: WebSocket monitoring logs showing TP and SL order placement (Image 5)*

📸 *Screenshot 7: Binance UI showing both OCO orders active (Image 7)*

Outcome: The WebSocket successfully monitored both orders in real-time. When the Take Profit filled at $89,840, the system automatically cancelled the Stop Loss order, demonstrating proper OCO bracket behavior.

> ⚠️ **Important Note:** Client-side OCO management introduces risks compared to server-side execution. During stress testing, manual interruption (Ctrl+C) sometimes caused race conditions. The implementation includes proper cleanup handlers to mitigate this.

4.2 Grid Trading Strategy

Concept: Place a ladder of buy and sell orders within a price range to profit from market volatility without predicting direction.

> **Command:**
> `PYTHONPATH=src uv run python src/advanced/grid.py BTCUSDT 89500 90500 5 0.002`

Grid Configuration:

Parameter	Value	Description
Lower Bound	$89,500	Minimum price for buy orders
Upper Bound	$90,500	Maximum price for sell orders
Grid Levels	5	Number of price levels
Quantity/Level	0.002 BTC	Order size at each level

Execution Logic:

- Fetch current market price: ~$90,000
- Calculate price step: ($90,500 - $89,500) / 5 = $200
- Place BUY orders below current price (levels at $89,500, $89,700, $89,900)
- Place SELL orders above current price (levels at $90,100, $90,300, $90,500)

- Skip orders too close to current price (within 10% of step size)

Result: Successfully placed 4-5 grid orders with proper spacing. The bot automatically calculated precision requirements from exchange info and formatted prices accordingly.

4.3 TWAP (Time-Weighted Average Price)

Purpose: Split large orders into smaller chunks executed over time to minimize market impact and avoid adverse price movement.

> **Command:**
> `PYTHONPATH=src uv run python src/advanced/twap.py BTCUSDT BUY 0.01 5 3`

Execution Plan:

- Total Quantity: 0.01 BTC
- Duration: 5 minutes
- Chunks: 3
- Chunk Size: 0.00333 BTC each
- Interval: 100 seconds between executions

Implementation Features:

- Dynamic precision adjustment based on exchange symbol info
- Remaining quantity correction on final chunk to handle rounding
- Graceful interruption handling with partial execution logging
- Detailed progress tracking with chunk numbering

5. Challenges & Solutions

Challenge	Root Cause	Solution Implemented
Order Notional < 100	Testnet requires minimum order value ~$100 (vs $5 on	Adjusted test quantities to 0.002-0.005 BTC (~$180-

	Mainnet)	$450) to pass validation
Stop Price Rejection	Stop price placed on wrong side of current market (e.g., stop buy below market)	Added direction validation: SELL stops must be below market, BUY stops above
ModuleNotFoundError	Python failing to find utils module from advanced/ subdirectory	Used PYTHONPATH=src environment variable for advanced scripts
WebSocket Connection Stability	Network interruptions or testnet instability	Implemented reconnection logic and graceful degradation with error logging
Precision Errors	Hardcoded decimal places not matching exchange requirements	Dynamic precision fetching via exchange_info() API endpoint

6. Logging & Observability

6.1 Dual-Channel Logging System

- **Console (INFO level):** User-friendly progress updates and confirmations
- **bot.log (DEBUG level):** Complete API request/response traces for debugging

6.2 Log Structure

Each log entry includes:

- Timestamp (millisecond precision)
- Log Level (INFO, DEBUG, ERROR)
- Source file and line number
- Descriptive message with context

```
2026-01-21 21:26:36,623 - INFO - [limit_orders.py:26] - Initiating
LIMIT BUY order for 0.005 BTCUSDT @ 90500.0 2026-01-21 21:26:37,070 -
INFO - [limit_orders.py:37] - Order Placed: ID 11854144928 | Status:
NEW DEBUG:limit_order:Full API Response: {'orderId': 11854144928,
'symbol': 'BTCUSDT', ...}
```

📸 *Screenshot 8: Trade History showing all executed orders with timestamps and fees (Image 9)*

7. Testing Methodology

7.1 Test Environment

- **Platform:** Binance Futures Testnet (testnet.binancefuture.com)
- **Test Funds:** $100,000 USDT (simulated)
- **Trading Pair:** BTCUSDT Perpetual
- **Date Range:** January 21, 2026

7.2 Verification Steps

1. Execute command via CLI
2. Monitor terminal output for immediate feedback
3. Verify order placement in bot.log with full API response
4. Check Binance Testnet UI for order status
5. Confirm fills in Order History and Trade History

8. Key Learnings & Best Practices

8.1 API Design Patterns

- Always fetch symbol precision dynamically rather than hardcoding
- Implement exponential backoff for rate limit handling

- Use WebSocket user data streams for real-time order monitoring
- Validate all inputs before API calls to reduce error rates

8.2 Production Considerations

✅ **Implemented:**

- Comprehensive error handling with detailed logging
- Input validation preventing invalid API calls
- Graceful degradation when WebSocket connections fail
- Environment-based configuration (Testnet vs Production)

⚠️ **Future Enhancements:**

- Database persistence for order history
- Position management and P&L tracking
- Risk management (max position size, daily loss limits)
- Backtesting framework for strategy validation

9. Conclusion

This project successfully implements a production-ready trading bot meeting all mandatory requirements (Market and Limit orders) while delivering three advanced strategies that demonstrate sophisticated algorithmic trading concepts.

Achievements:

- ✅ Core order types with robust validation
- ✅ OCO implementation using WebSocket real-time monitoring
- ✅ Grid trading for range-bound market conditions
- ✅ TWAP algorithm for large order execution
- ✅ Professional logging and error handling

- ✅ Comprehensive documentation (README.md + this report)

The system demonstrates enterprise-grade code organization, type safety, and operational awareness through detailed logging. While tested on Binance Testnet, the architecture is production-ready with environment-based configuration enabling seamless transition to live trading with appropriate risk management controls.

Repository: github.com/ramansingh/raman-binance-bot

Contact: gillramansingh900@gmail.com