BINANCE FUTURES TESTNET TRADING BOT — FINAL PROJECT REPORT
Author: Harinath
Platform: Ubuntu Linux (VS Code)
Exchange: Binance USDT-M Futures Testnet

---
1. INTRODUCTION

This project is a complete command-line trading bot built in Python for the Binance USDT-M Futures Testnet. It provides reliable and production-style support for multiple order types including Market orders, Limit orders, Stop-Market, Stop-Limit, OCO (Take Profit + Stop Loss), and TWAP (Time Weighted Average Price).

The objective was to build a secure, modular, scalable, and validated trading system that uses Binance REST APIs, supports HMAC-SHA256 signature authentication, performs exchange filter validation, logs every action, and handles errors gracefully.

This report explains the architecture, features, challenges, testing, and outcomes of the project.

---
2. PROJECT OBJECTIVES

Mandatory Requirements:
1. Connect to Binance Futures Testnet using secure REST APIs.
2. Implement HMAC-SHA256 signing for all private endpoints.
3. Support Market and Limit orders.
4. Implement clean error handling and validations.
5. Provide a proper CLI interface for all actions.
6. Log all requests, responses, and errors in bot.log.
7. Load API keys from a .env configuration file.
8. Use a modular folder structure.

Advanced Features Completed:
1. OCO (Take Profit + Stop Loss)
2. Stop-Market Order
3. Stop-Limit Order
4. Exchange filters (tick size, step size, minNotional, percent price)
5. Cancel order + Cancel all orders
6. Inspect open positions
7. Structured detailed logging
8. TWAP: Time Weighted Average Price (Bonus)

The project successfully implements all mandatory and advanced features.

---
3. SYSTEM ARCHITECTURE

The bot follows a modular, layered architecture. Each responsibility is isolated into a dedicated Python module.

cli.py:
Handles all command-line Commands. Uses the Click framework. Dispatches commands to modules such as market orders, limit orders, stop orders, OCO, TWAP, etc.

client_wrapper.py:
Implements REST API communication. Handles signing, request sending, GET/POST methods, syncing server time, API headers, HMAC-SHA256 signature, and error handling.

utils.py:
Contains helper functions for exchange filters, step size, tick size rounding, percent price validation, and general utilities.

logger.py:
Configures Loguru to write detailed logs into bot.log. Logs include request URLs, parameters, responses, errors, OCO events, Stop orders, TWAP executions.

market_orders.py:
Implements Market BUY and SELL orders.

limit_orders.py:
Implements Limit BUY and SELL with filter validation.

advanced/stop_orders.py:
Implements both Stop-Market and Stop-Limit orders.
Prevents Binance error: “Order would immediately trigger”.
Validates trigger side based on current market price.

advanced/oco.py:
Implements OCO brackets including entry order, take-profit leg, stop-loss leg, detached mode, and waiting mode.

advanced/twap.py:
Implements TWAP slicing logic. Splits a large order into smaller slices and executes them after fixed intervals using Market orders.

Design Principles Used:
Modular coding
Separation of concerns
Error isolation
Full logging for transparency
Secure API handling
Scalable structure for future enhancements

---

4. PROJECT FOLDER STRUCTURE

```
harinath_binance_bot/
    src/
        cli.py
        config.py
        client_wrapper.py
        utils.py
        market_orders.py
        limit_orders.py
        logger.py
        advanced/
            stop_orders.py
            oco.py
            twap.py
    .env
    .env.example
    requirements.txt
    README.md
    report.pdf
    bot.log
    .gitignore
```

---

5. FEATURE SUMMARY

Environment and Configuration:
- API keys loaded securely through .env
- No hardcoded secrets
- .gitignore protects sensitive files

REST API Integration:
- Uses Requests library
- HMAC-SHA256 signing with proper timestamp and recvWindow
- Automatically syncs server time on each request
- Full parameter logging before sending to Binance

Market Orders:
- Market BUY and SELL implemented
- Quantities validated and rounded

Limit Orders:
- Comprehensive validation including tick size, step size, minNotional, market boundaries, and percent price rules

OCO Orders:
- Entry order + Take Profit + Stop Loss
- Supports detached mode (return immediately)

- Supports waiting mode (polling entry fill)

Stop-Limit Orders:
- stopPrice triggers creation of a limit post-only style order
- Protected against “Order would immediately trigger”

Stop-Market Orders:
- stopPrice triggers a market order
- Supports closePosition functionality

TWAP Orders (Bonus):
- Splits a large order into “parts”
- Executes each slice after a fixed interval
- All slices use MARKET execution to guarantee fill

Order Management:
- View open orders
- Cancel specific order (orderId or clientOrderId)
- Cancel all orders with safe handling for unknown orders

Position Inspection:
- Prints symbol, position amount, entry price, and unrealized profit

Logging:
All actions are stored in bot.log including:
REST URLs
Parameters
API responses
Errors
Validation details
OCO execution flow
STOP orders
TWAP slices

---
6. MAJOR CHALLENGES AND SOLUTIONS
---

1. .env Loading Failure:
Cause: Inline Python lost context for dotenv.
Solution: Always use load_dotenv(dotenv_path=".env").

2. Invalid API Keys:
Initial testnet keys expired.
Solution: Regenerated new keys.

3. Parameter Signing Errors:
Early attempts failed due to incorrect query-string formatting.

Solution: Use standard urlencode() before signing.

4. Min Notional Rejections:
Small quantities like 0.001 BTC rejected.
Solution: Increased to 0.002 BTC and added filter validation.

5. “Order Would Immediately Trigger” (-2021) Error:
Stop-Limit logic violated trigger rules.
Solution: Added market-price check:
BUY stopPrice must be greater than current price.
SELL stopPrice must be lower than current price.

6. Tick Size Rounding:
Invalid price rejected by Binance.
Solution: Implemented tick-size rounding in utils.py.

7. Cancel Order Errors:
Unknown or already closed order returned -2011.
Solution: Treat these errors as “already cancelled”.

8. OCO Timing Issues:
Detached mode worked but waiting mode needed polling.
Solution: Added entry fill polling loop.

9. TWAP Interval Trigger Inaccuracy:
Needed precise delay between slices.
Solution: Added safe sleep handling and logging.

---
7. TEST RESULTS

Environment Used:
Ubuntu Linux
Python 3.8
VS Code
Binance USDT-M Testnet

Successful Tests:
auth-check confirmed API access
Fetched exchange filters successfully
Market Buy Order Worked
Market Sell Order Worked
Limit Buy and Sell orders placed correctly
Open orders displayed correctly
Cancel specific order worked
Cancel-all processed all open orders
Position inspection displayed correct entries
OCO orders tested in both modes

Stop-Limit orders validated and executed
Stop-Market orders executed properly
TWAP tested with 10 slices over intervals
bot.log captured all operations

All functions passed testing with valid Binance responses.

---
8. SUGGESTED SCREENSHOTS FOR THE PDF
---
1. Successful auth-check
2. Market order placed
3. Limit order placed
4. Output of open orders
5. Output of cancel-all
6. OCO entry placement
7. STOP-LIMIT success message
8. STOP-MARKET success message
9. inspect-pos output
10. tail -n 40 bot.log

---
9. CONCLUSION

This project fulfills all assignment requirements completely, including advanced features.

The system is secure, modular, scalable, and production-grade. It supports:
Multiple order types
Stop and OCO workflows
Exchange filter validation
Time-based execution via TWAP
Comprehensive logging
Command-line automation

This project demonstrates strong understanding of:
Python API integration
Trading systems
Error handling
Order validation
Modular architecture
Testnet trading operations

---
10. FUTURE ENHANCEMENTS

Websocket live price streaming
Trailing Stop-Loss and Take-Profit
Full automated strategy engine

Grid trading automation
Hedge mode support
Real-time liquidation monitoring
Telegram notification integration
Portfolio PnL dashboard

END OF REPORT
