FlexiRule Framework Documentation

1. RequirementsThe FlexiRule framework is designed to be a highly flexible, scalable, and extensible rule-based strategy engine for the ANQuant Trading application. It must support any mathematically expressible trading strategy, including SMC, technical indicators, price action, harmonics, and wave theory, while integrating seamlessly with the existing NoRust architecture.1.1 Functional Requirements

1. Flexible Strategy Definition:
   * Support any trading strategy expressible through mathematical conditions (e.g., SMC order blocks, RSI crossovers, bullish engulfing patterns).
   * Use a YAML-based configuration to define strategies, including entry rules, exit rules, stop-loss, targets, indicators, and patterns.
   * Allow multiple strategies to run concurrently for stocks in watchlists (e.g., config/markets/india/watchlists/smc.yaml).
2. Rule Evaluation:
   * Evaluate complex conditions with operators (>, <, =, >=, <=, !=, and, or) on OHLCV fields (open, close, high, low, volume), indicators, and custom patterns.
   * Support multi-timeframe analysis (e.g., 5-minute for bias, 1-minute for confirmation).
   * Generate trading signals (BUY, SELL, HOLD, PARTIAL\_SELL) based on rule outcomes.
3. Stop-Loss and Target Flexibility:
   * Support fixed percentage stop-loss and targets (e.g., 1.5% stop, 5% target).
   * Allow indicator-based trailing stops (e.g., trail at 2 \* ATR\_14 or SMA\_20).
   * Enable partial exits (e.g., 50% at 5% gain, 25% at 8% gain) with configurable percentages.
   * Support condition-based exits (e.g., rsi > 70, bearish\_choch = true).
   * Allow breakeven adjustments (e.g., move stop to entry after 3:1 risk-reward).
   * Provide volatility-based stops (e.g., ATR multiples) and time-based exits (e.g., exit after 30 minutes).
4. Pattern and Indicator Support:
   * Support standard indicators (e.g., RSI, SMA, Bollinger Bands, ATR) computed by IndicatorEngine (src/rs/indicator/src/engine.rs).
   * Allow custom indicators defined via formulas (e.g., VWAP as (sum(close \* volume) / sum(volume))).
   * Support custom patterns (e.g., SMC order blocks, liquidity pools, equal highs/lows) with configurable criteria and lookback periods.
   * Handle multi-candle patterns (e.g., equal highs over 3 candles).
5. Integration with ANQuant Architecture:
   * Consume real-time OHLCV data from Kafka (ohlcv\_1min, ohlcv\_5min).
   * Retrieve indicators from Redis (symbol:indicators:1min, symbol:indicators:5min).
   * Fetch historical OHLCV from Redis (symbol:ohlcv:1min) or PostgreSQL (ohlcv table).
   * Publish trading signals to Kafka (signals) and Redis (signals:<strategy\_name>).
   * Support multi-market trading (India, USA) using watchlists (e.g., config/markets/india/watchlists/meanhunter.yaml).
   * Adjust for corporate actions (e.g., SUNPHARMA dividend ₹5.50) via src/py/core/historical\_data\_manager.py.
6. Trade Management:
   * Support position scaling (e.g., add to position if rsi < 20).
   * Allow dynamic risk-reward ratios based on market conditions (e.g., 5:1 based on ATR).
   * Track position states (entry price, partial exits, breakeven status) for each symbol.
7. Extensibility:
   * Allow new strategies to be added via YAML files without modifying core code.
   * Support custom pattern detection (e.g., SMC-specific patterns like CHoCH, BOS).
   * Enable easy integration of new indicators via IndicatorEngine.

1.2 Non-Functional Requirements

1. Performance:
   * Process OHLCV data in real-time with low latency (<100ms per symbol evaluation).
   * Handle up to 100 stocks per watchlist (e.g., config/markets/india/watchlists/smc.yaml) and multiple strategies concurrently.
   * Cache frequently accessed data (e.g., indicators, patterns) in Redis for fast retrieval.
2. Scalability:
   * Scale to support thousands of symbols across multiple markets (India, USA).
   * Handle multiple timeframes (1min, 5min, 30min) simultaneously.
3. Reliability:
   * Ensure robust error handling and logging to logs/strategy/ for auditability (e.g., SEBI compliance).
   * Maintain data consistency across Redis, Kafka, and PostgreSQL.
4. Security:
   * Use a safe expression parser (e.g., asteval) for rule evaluation to prevent code injection risks.
   * Restrict access to configuration files and logs via file permissions.
5. Maintainability:
   * Modular design with clear separation of concerns (e.g., StrategyManager for orchestration, RuleEngine for evaluation).
   * Comprehensive test suite in tests/test\_strategy\_manager.py.
6. Compatibility:
   * Integrate with existing Rust-based components (MarketDataEngine, IndicatorEngine).
   * Support brokers (e.g., AngelOne, Fyers, Zerodha) via symbol mappings (e.g., config/markets/india/brokers/symbol\_mappings/angelone.yaml).

2. Proposed DesignThe FlexiRule framework is implemented as a Python-based rule engine within the ANQuant architecture, with components in src/py/core/strategy/ and configurations in config/markets/india/strategies/. It consists of:

* StrategyManager (src/py/core/strategy/manager.py): Orchestrates multiple strategies, loads YAML configurations, and processes OHLCV data from Kafka.
* RuleEngine (src/py/core/strategy/rules.py): Evaluates rules, patterns, and stop-loss/target conditions, generating trading signals.
* Integration Components:
  + Consumes OHLCV from Kafka (ohlcv\_1min, ohlcv\_5min).
  + Retrieves indicators from Redis (symbol:indicators:1min).
  + Stores historical data in PostgreSQL (ohlcv table).
  + Publishes signals to Kafka (signals) and Redis (signals:<strategy\_name>).
* Configuration: YAML files define strategies, indicators, patterns, and trade management rules.

2.1 Architecture Overview

* Input:
  + Real-time OHLCV from Kafka (ohlcv\_1min, ohlcv\_5min) via MarketDataEngine (src/rs/market\_data/src/engine.rs).
  + Indicators from Redis (symbol:indicators:1min) via IndicatorEngine (src/rs/indicator/src/engine.rs).
  + Historical OHLCV from Redis (symbol:ohlcv:1min) or PostgreSQL (ohlcv table).
* Processing:
  + StrategyManager loads strategy YAMLs from config/markets/india/strategies/.
  + RuleEngine evaluates entry/exit rules, patterns, stop-loss, and targets for each symbol.
* Output:
  + Signals (BUY, SELL, HOLD, PARTIAL\_SELL) published to Kafka (signals) and Redis (signals:<strategy\_name>).
  + Logs written to logs/strategy/<strategy\_name>\_2025-07-18.log for auditability.

2.2 Example Strategy: Advanced SMC StrategyTo demonstrate FlexiRule’s capabilities, we’ll use an Advanced SMC Strategy that incorporates:

* Market Structure: Identifies bearish/bullish trends using swing points, Break of Structure (BOS), and Change of Character (CHoCH).
* Order Blocks: Detects institutional intention via buy-to-sell ranges.
* Liquidity Pools: Identifies Asia session ranges or equal highs/lows for manipulation.
* Multi-Timeframe Confirmation: Uses 1-minute timeframe for entry confirmation within a 5-minute bias.
* Flexible Stop-Loss/Target:
  + Fixed 1.5% stop-loss initially.
  + Trail stop using 2 \* ATR\_14 after breakeven.
  + Partial exit of 50% at 5% gain, 25% at 8% gain.
  + Exit remaining position if rsi > 70 or bearish\_choch = true.
  + Move stop to breakeven after 3:1 risk-reward.

2.2.1 YAML ConfigurationFile: config/markets/india/strategies/advanced\_smc.yaml

yaml

name: advanced\_smc

timeframe: 5min

watchlist: config/markets/india/watchlists/smc.yaml

indicators:

- name: rsi

type: rsi

period: 14

- name: atr\_14

type: atr

period: 14

- name: sma\_20

type: sma

period: 20

- name: vwap

type: custom

formula: "(sum(close \* volume) / sum(volume)) over 20 periods"

patterns:

- name: market\_structure

type: smc

lookback: 50

criteria: "close > prev\_swing\_high and prev\_swing\_high > prev\_prev\_swing\_high ? 'bullish' : (close < prev\_swing\_low and prev\_swing\_low < prev\_prev\_swing\_low ? 'bearish' : 'ranging')"

- name: bullish\_order\_block

type: smc

lookback: 20

criteria: "high > prev\_high and low < prev\_low and volume > avg\_volume\_20"

- name: bearish\_order\_block

type: smc

lookback: 20

criteria: "low < prev\_low and high > prev\_high and volume > avg\_volume\_20"

- name: liquidity\_pool

type: smc

lookback: 10

criteria: "abs(high - low) < 0.5 \* atr\_14 and timestamp >= '02:30:00' and timestamp <= '05:30:00'"

- name: eqh

type: smc

lookback: 5

criteria: "abs(high - prev\_high) < 0.1 \* atr\_14 for 3 candles"

- name: bullish\_bos

type: smc

lookback: 20

criteria: "close > prev\_swing\_high and market\_structure = bullish"

- name: bearish\_bos

type: smc

lookback: 20

criteria: "close < prev\_swing\_low and market\_structure = bearish"

- name: bullish\_choch

type: smc

lookback: 20

criteria: "close < prev\_higher\_low and market\_structure = bullish"

- name: bearish\_choch

type: smc

lookback: 20

criteria: "close > prev\_lower\_high and market\_structure = bearish"

- name: bearish\_confirmation\_1min

type: smc

timeframe: 1min

lookback: 5

criteria: "close < prev\_low and prev\_high < prev\_prev\_high"

entry\_rules:

- condition: "market\_structure = bearish and bearish\_order\_block = true and liquidity\_pool = true and bearish\_confirmation\_1min = true"

weight: 0.8

- condition: "rsi < 30"

weight: 0.2

exit\_rules:

- condition: "bullish\_choch = true or rsi > 70"

weight: 0.7

- condition: "close > bearish\_order\_block\_high"

weight: 0.3

stop\_loss:

type: multi

rules:

- type: fixed

value: 1.5% *# Initial stop-loss at 1.5% below entry*

- type: indicator

indicator: atr\_14

multiplier: 2.0 *# Trail at 2 \* ATR\_14 below highest price*

after: breakeven *# Apply after breakeven trigger*

- type: condition

condition: "close < sma\_20" *# Exit if price falls below SMA\_20*

after: partial\_exit\_1 *# Apply after first partial exit*

target:

type: multi

rules:

- type: fixed

value: 5% *# First target at 5% above entry*

partial\_exit: 50% *# Exit 50% of position*

id: partial\_exit\_1

- type: fixed

value: 8% *# Second target at 8% above entry*

partial\_exit: 25% *# Exit 25% of position*

id: partial\_exit\_2

- type: indicator

indicator: sma\_20

trail: true *# Trail target at SMA\_20 for remaining position*

- type: condition

condition: "rsi > 70 or bearish\_choch = true" *# Exit remaining position*

trade\_management:

breakeven:

trigger: 3 *# Move stop-loss to breakeven after 3:1 risk-reward*

partial\_exits:

- id: partial\_exit\_1

level: 5% *# 5% above entry*

percentage: 50% *# Exit 50% of position*

- id: partial\_exit\_2

level: 8% *# 8% above entry*

percentage: 25% *# Exit 25% of position*

scale\_in:

- condition: "rsi < 20 and close > entry\_price \* 1.02"

percentage: 25% *# Add 25% to position if RSI < 20 and price is 2% above entry*

2.2.2 Strategy ExplanationThis Advanced SMC Strategy demonstrates FlexiRule’s capabilities:

* Market Structure: Uses market\_structure, bullish\_bos, bearish\_bos, bullish\_choch, and bearish\_choch to identify trends and reversals.
* Order Blocks: Detects bullish\_order\_block and bearish\_order\_block as institutional intention zones.
* Liquidity Pools: Identifies liquidity\_pool during Asia session (2:30 AM–5:30 AM IST) and eqh for equal highs.
* Multi-Timeframe Confirmation: Uses bearish\_confirmation\_1min to confirm entries on the 1-minute timeframe.
* Stop-Loss:
  + Initial fixed stop at 1.5% below entry.
  + Trails at 2 \* ATR\_14 after breakeven.
  + Exits if close < sma\_20 after the first partial exit.
* Target:
  + Exits 50% at 5% gain, 25% at 8% gain.
  + Trails remaining position at SMA\_20.
  + Exits if rsi > 70 or bearish\_choch = true.
* Trade Management:
  + Moves stop to breakeven after 3:1 risk-reward.
  + Scales into position by 25% if rsi < 20 and price is 2% above entry.

2.3 Implementation Details2.3.1 StrategyManager (src/py/core/strategy/manager.py)The StrategyManager orchestrates multiple strategies, loading YAML files and delegating rule evaluation to RuleEngine.

python

*# src/py/core/strategy/manager.py*

from src.py.util.logging import setup\_logging

from src.py.messaging.kafka\_client import KafkaClient

from src.py.messaging.redis\_client import RedisClient

from src.py.core.strategy.rules import RuleEngine

import asyncio

from typing import Dict, Any

import json

import yaml

import os

logger = setup\_logging("strategy\_manager", log\_type="strategy")

class StrategyManager:

def \_\_init\_\_(self, config: Dict[str, Any], redis\_client: RedisClient):

self.config = config

self.redis\_client = redis\_client

self.kafka\_client = KafkaClient(config["global"]["kafka"]) if not config['global'].get('offline\_mode', False) else None

self.strategies = self.\_load\_strategy\_configs()

self.rule\_engines = {strategy['name']: RuleEngine(strategy, redis\_client, config) for strategy in self.strategies}

logger.debug(f"Initialized StrategyManager with {len(self.strategies)} strategies")

def \_load\_strategy\_configs(self):

"""Load all strategy YAML files from config/markets/india/strategies/."""

strategies = []

strategy\_dir = os.path.join(os.path.dirname(\_\_file\_\_), "../../../../config/markets/india/strategies")

try:

for strategy\_file in os.listdir(strategy\_dir):

if strategy\_file.endswith(('.yaml', '.yml')):

with open(os.path.join(strategy\_dir, strategy\_file), 'r', encoding='utf-8') as f:

strategy\_config = yaml.safe\_load(f)

if strategy\_config:

strategies.append(strategy\_config)

logger.debug(f"Loaded strategy config: {strategy\_file}")

except Exception as e:

logger.error(f"Failed to load strategy configs from {strategy\_dir}: {e}")

raise

return strategies

async def initialize(self):

logger.info("Initializing StrategyManager")

try:

if self.kafka\_client:

self.kafka\_client.connect()

topics = list(set(f"ohlcv\_{strategy['timeframe']}" for strategy in self.strategies))

for strategy in self.strategies:

for pattern in strategy.get('patterns', []):

if pattern.get('timeframe'):

topics.append(f"ohlcv\_{pattern['timeframe']}")

self.kafka\_client.subscribe(topics)

logger.debug(f"Subscribed to Kafka topics: {topics}")

await self.redis\_client.redis.ping()

logger.debug("Verified Redis connectivity")

for name, rule\_engine in self.rule\_engines.items():

await rule\_engine.initialize()

logger.debug(f"Initialized RuleEngine for strategy {name}")

logger.info("StrategyManager initialized")

except Exception as e:

logger.error(f"Failed to initialize StrategyManager: {e}")

raise

async def start(self):

logger.info("Starting StrategyManager")

if not self.config['global'].get('offline\_mode', False):

asyncio.create\_task(self.\_process\_messages())

logger.info("StrategyManager started")

async def \_process\_messages(self):

try:

while True:

msg = self.kafka\_client.poll(timeout=1.0)

if msg:

ohlcv = msg['value']

symbol = ohlcv['tradingsymbol']

topic = msg['topic']

timeframe = topic.replace('ohlcv\_', '')

logger.debug(f"Processing OHLCV for {symbol} on {timeframe} timeframe")

await self.\_process\_strategy\_signals(symbol, timeframe, ohlcv)

await asyncio.sleep(0.1)

except Exception as e:

logger.error(f"Error in strategy message processing loop: {e}", exc\_info=True)

raise

async def \_process\_strategy\_signals(self, symbol: str, timeframe: str, ohlcv: Dict):

try:

indicators\_data = await self.redis\_client.get(f"{symbol}:indicators:{timeframe}")

indicators = indicators\_data if isinstance(indicators\_data, dict) else indicators\_data[0] if indicators\_data else {}

for strategy in self.strategies:

if symbol in self.\_load\_watchlist(strategy['watchlist']):

if strategy['timeframe'] == timeframe or any(pattern.get('timeframe') == timeframe for pattern in strategy.get('patterns', [])):

rule\_engine = self.rule\_engines[strategy['name']]

signal = await rule\_engine.evaluate(symbol, ohlcv, indicators)

if signal and signal != "HOLD":

signal\_data = {

'symbol': symbol,

'signal': signal,

'timestamp': ohlcv['timestamp'],

'price': ohlcv['close'],

'strategy': strategy['name']

}

await self.redis\_client.publish(f"signals:{strategy['name']}", json.dumps(signal\_data))

self.kafka\_client.produce("signals", key=symbol, value=signal\_data)

logger.info(f"Generated signal for {symbol}: {signal} (strategy: {strategy['name']})")

except Exception as e:

logger.error(f"Error processing strategy signals for {symbol}:{timeframe}: {e}", exc\_info=True)

def \_load\_watchlist(self, watchlist\_path: str) -> set:

"""Load watchlist symbols from YAML file."""

try:

with open(os.path.join(os.path.dirname(\_\_file\_\_), f"../../../../{watchlist\_path}"), 'r', encoding='utf-8') as f:

watchlist = yaml.safe\_load(f)

return {item['tradingsymbol'] for item in watchlist}

except Exception as e:

logger.error(f"Failed to load watchlist {watchlist\_path}: {e}")

return set()

2.3.2 RuleEngine (src/py/core/strategy/rules.py)The RuleEngine evaluates rules, patterns, stop-loss, and targets, supporting multi-timeframe analysis and flexible trade management.

python

*# src/py/core/strategy/rules.py*

from src.py.util.logging import setup\_logging

from src.py.messaging.redis\_client import RedisClient

import asyncio

from typing import Dict, Any, Optional, List, Tuple

import operator

import pandas\_ta as pta

import pandas as pd

import numpy as np

from asteval import Interpreter *# Safer alternative to eval*

logger = setup\_logging("rules", log\_type="strategy")

class RuleEngine:

def \_\_init\_\_(self, strategy\_config: Dict[str, Any], redis\_client: RedisClient, config: Dict[str, Any]):

self.strategy\_config = strategy\_config

self.name = strategy\_config['name']

self.timeframe = strategy\_config['timeframe']

self.indicators = strategy\_config.get('indicators', [])

self.patterns = strategy\_config.get('patterns', [])

self.entry\_rules = strategy\_config.get('entry\_rules', [])

self.exit\_rules = strategy\_config.get('exit\_rules', [])

self.stop\_loss = strategy\_config.get('stop\_loss', {})

self.target = strategy\_config.get('target', {})

self.trade\_management = strategy\_config.get('trade\_management', {})

self.redis\_client = redis\_client

self.config = config

self.positions = {} *# symbol: {entry\_price, highest\_price, lowest\_price, partial\_exits, breakeven\_set}*

self.pattern\_cache = {} *# symbol: {pattern\_name: bool/value}*

self.asteval = Interpreter() *# Safe expression evaluator*

async def initialize(self):

logger.info(f"Initializing RuleEngine for strategy {self.name}")

try:

await self.redis\_client.redis.ping()

for rule in self.entry\_rules + self.exit\_rules:

if 'condition' not in rule or 'weight' not in rule:

raise ValueError(f"Invalid rule format in strategy {self.name}: {rule}")

for indicator in self.indicators:

if 'type' not in indicator or 'name' not in indicator:

raise ValueError(f"Invalid indicator format in strategy {self.name}: {indicator}")

for pattern in self.patterns:

if 'type' not in pattern or 'name' not in pattern:

raise ValueError(f"Invalid pattern format in strategy {self.name}: {pattern}")

logger.info(f"RuleEngine for {self.name} initialized")

except Exception as e:

logger.error(f"Failed to initialize RuleEngine for {self.name}: {e}")

raise

async def evaluate(self, symbol: str, ohlcv: Dict, indicators: Dict) -> Optional[str]:

"""Evaluate rules and return signal (BUY, SELL, HOLD, PARTIAL\_SELL)."""

try:

position = self.positions.get(symbol, {

'entry\_price': None,

'highest\_price': None,

'lowest\_price': None,

'partial\_exits': {},

'breakeven\_set': False

})

close = ohlcv['close']

*# Update pattern cache*

await self.\_update\_patterns(symbol, ohlcv)

*# Check stop-loss and target for open position*

if position['entry\_price'] is not None:

sl\_signal = self.\_evaluate\_stop\_loss(symbol, close, position, indicators)

if sl\_signal:

self.positions.pop(symbol, None)

return sl\_signal

target\_signal = self.\_evaluate\_target(symbol, close, position, indicators)

if target\_signal:

if target\_signal == "PARTIAL\_SELL":

return None *# Partial exit, keep position open*

self.positions.pop(symbol, None)

return target\_signal

exit\_signal = self.\_evaluate\_rules(self.exit\_rules, ohlcv, indicators)

if exit\_signal:

self.positions.pop(symbol, None)

return "SELL"

*# Check entry rules*

entry\_signal = self.\_evaluate\_rules(self.entry\_rules, ohlcv, indicators)

if entry\_signal:

self.positions[symbol] = {

'entry\_price': close,

'highest\_price': close,

'lowest\_price': close,

'partial\_exits': {},

'breakeven\_set': False

}

return "BUY"

*# Check scale-in conditions*

if position['entry\_price'] is not None:

scale\_in\_signal = self.\_evaluate\_scale\_in(symbol, close, position, indicators)

if scale\_in\_signal:

return "BUY" *# Additional position*

return "HOLD"

except Exception as e:

logger.error(f"Error evaluating rules for {symbol} in {self.name}: {e}", exc\_info=True)

return "HOLD"

async def \_update\_patterns(self, symbol: str, ohlcv: Dict):

"""Update pattern cache for the symbol."""

try:

for pattern in self.patterns:

pattern\_name = pattern['name']

pattern\_type = pattern['type']

lookback = pattern.get('lookback', 20)

criteria = pattern.get('criteria', '')

timeframe = pattern.get('timeframe', self.timeframe)

historical = await self.redis\_client.get(f"{symbol}:ohlcv:{timeframe}")

if not historical or len(historical) < lookback:

logger.debug(f"Insufficient historical data for pattern {pattern\_name} on {symbol}")

self.pattern\_cache.setdefault(symbol, {})[pattern\_name] = False

continue

df = pd.DataFrame(historical[-lookback:])

if pattern\_type == 'smc':

self.pattern\_cache.setdefault(symbol, {})[pattern\_name] = self.\_evaluate\_smc\_pattern(df, criteria)

elif pattern\_type == 'price\_action':

self.pattern\_cache.setdefault(symbol, {})[pattern\_name] = self.\_evaluate\_price\_action\_pattern(df, criteria)

elif pattern\_type == 'custom':

self.pattern\_cache.setdefault(symbol, {})[pattern\_name] = self.\_evaluate\_custom\_pattern(df, criteria)

else:

logger.error(f"Unsupported pattern type {pattern\_type} for {pattern\_name}")

self.pattern\_cache.setdefault(symbol, {})[pattern\_name] = False

except Exception as e:

logger.error(f"Error updating patterns for {symbol}: {e}")

self.pattern\_cache.setdefault(symbol, {})[pattern\_name] = False

def \_evaluate\_smc\_pattern(self, df: pd.DataFrame, criteria: str) -> Any:

"""Evaluate SMC patterns (e.g., order blocks, market structure)."""

try:

locals\_dict = {

'close': df['close'].iloc[-1],

'high': df['high'].iloc[-1],

'low': df['low'].iloc[-1],

'volume': df['volume'].iloc[-1],

'prev\_high': df['high'].shift(1).iloc[-1],

'prev\_low': df['low'].shift(1).iloc[-1],

'prev\_close': df['close'].shift(1).iloc[-1],

'avg\_volume\_20': df['volume'].rolling(20).mean().iloc[-1],

'atr\_14': pta.atr(df['high'], df['low'], df['close'], 14).iloc[-1],

'timestamp': pd.to\_datetime(df['timestamp'].iloc[-1]).strftime('%H:%M:%S')

}

*# Add swing points*

swings = self.\_detect\_swing\_points(df, 20)

locals\_dict.update({

'prev\_swing\_high': swings['highs'][-1][1] if swings['highs'] else 0.0,

'prev\_swing\_low': swings['lows'][-1][1] if swings['lows'] else 0.0,

'prev\_prev\_swing\_high': swings['highs'][-2][1] if len(swings['highs']) > 1 else 0.0,

'prev\_prev\_swing\_low': swings['lows'][-2][1] if len(swings['lows']) > 1 else 0.0,

'prev\_higher\_low': max(df['low'].iloc[-10:-1]) if len(df) > 1 else 0.0,

'prev\_lower\_high': min(df['high'].iloc[-10:-1]) if len(df) > 1 else 0.0

})

if '?' in criteria: *# Ternary-like expression for market\_structure*

condition, result = criteria.split('?')

true\_result, false\_result = result.split(':')

return self.asteval(condition.strip()) and true\_result.strip() or false\_result.strip()

return self.asteval(criteria, locals=locals\_dict)

except Exception as e:

logger.error(f"Error evaluating SMC pattern: {e}")

return False

def \_evaluate\_price\_action\_pattern(self, df: pd.DataFrame, criteria: str) -> bool:

"""Evaluate price action patterns (e.g., bullish engulfing)."""

try:

locals\_dict = {

'open': df['open'].iloc[-1],

'close': df['close'].iloc[-1],

'prev\_open': df['open'].shift(1).iloc[-1],

'prev\_close': df['close'].shift(1).iloc[-1],

'prev\_high': df['high'].shift(1).iloc[-1],

'prev\_low': df['low'].shift(1).iloc[-1]

}

return self.asteval(criteria, locals=locals\_dict)

except Exception as e:

logger.error(f"Error evaluating price action pattern: {e}")

return False

def \_evaluate\_custom\_pattern(self, df: pd.DataFrame, criteria: str) -> bool:

"""Evaluate custom patterns defined in YAML."""

try:

locals\_dict = {

'open': df['open'].iloc[-1],

'close': df['close'].iloc[-1],

'high': df['high'].iloc[-1],

'low': df['low'].iloc[-1],

'volume': df['volume'].iloc[-1],

'prev\_open': df['open'].shift(1).iloc[-1],

'prev\_close': df['close'].shift(1).iloc[-1],

'prev\_high': df['high'].shift(1).iloc[-1],

'prev\_low': df['low'].shift(1).iloc[-1],

'avg\_volume\_20': df['volume'].rolling(20).mean().iloc[-1],

'atr\_14': pta.atr(df['high'], df['low'], df['close'], 14).iloc[-1],

'timestamp': pd.to\_datetime(df['timestamp'].iloc[-1]).strftime('%H:%M:%S')

}

if 'for 3 candles' in criteria:

condition = criteria.replace('for 3 candles', '')

for i in range(-3, 0):

locals\_dict.update({

'high': df['high'].iloc[i],

'low': df['low'].iloc[i],

'timestamp': pd.to\_datetime(df['timestamp'].iloc[i]).strftime('%H:%M:%S')

})

if not self.asteval(condition, locals=locals\_dict):

return False

return True

return self.asteval(criteria, locals=locals\_dict)

except Exception as e:

logger.error(f"Error evaluating custom pattern: {e}")

return False

def \_detect\_swing\_points(self, df: pd.DataFrame, lookback: int) -> Dict:

"""Detect swing highs and lows."""

swings = {'highs': [], 'lows': []}

for i in range(2, min(len(df) - 2, lookback)):

if df['high'].iloc[i] > df['high'].iloc[i-1] and df['high'].iloc[i] > df['high'].iloc[i+1]:

swings['highs'].append((df['timestamp'].iloc[i], df['high'].iloc[i]))

if df['low'].iloc[i] < df['low'].iloc[i-1] and df['low'].iloc[i] < df['low'].iloc[i+1]:

swings['lows'].append((df['timestamp'].iloc[i], df['low'].iloc[i]))

return swings

def \_evaluate\_rules(self, rules: List[Dict], ohlcv: Dict, indicators: Dict) -> bool:

"""Evaluate a list of rules and return True if conditions are met."""

total\_weight = 0.0

required\_weight = sum(rule['weight'] for rule in rules) \* 0.75

for rule in rules:

condition = rule['condition']

weight = rule['weight']

if self.\_evaluate\_condition(condition, ohlcv, indicators):

total\_weight += weight

return total\_weight >= required\_weight

def \_evaluate\_condition(self, condition: str, ohlcv: Dict, indicators: Dict) -> bool:

"""Evaluate a complex condition with and/or operators."""

try:

*# Handle pattern-based conditions*

for pattern in self.patterns:

pattern\_name = pattern['name']

if f"{pattern\_name} = true" in condition.lower():

return self.pattern\_cache.get(ohlcv['tradingsymbol'], {}).get(pattern\_name, False)

if pattern\_name in condition and isinstance(self.pattern\_cache.get(ohlcv['tradingsymbol'], {}).get(pattern\_name), str):

self.asteval.symtable[pattern\_name] = self.pattern\_cache.get(ohlcv['tradingsymbol'], {}).get(pattern\_name)

if ' and ' in condition.lower():

conditions = condition.split(' and ')

return all(self.\_evaluate\_single\_condition(c.strip(), ohlcv, indicators) for c in conditions)

elif ' or ' in condition.lower():

conditions = condition.split(' or ')

return any(self.\_evaluate\_single\_condition(c.strip(), ohlcv, indicators) for c in conditions)

else:

return self.\_evaluate\_single\_condition(condition, ohlcv, indicators)

except Exception as e:

logger.error(f"Error evaluating condition {condition}: {e}")

return False

def \_evaluate\_single\_condition(self, condition: str, ohlcv: Dict, indicators: Dict) -> bool:

"""Evaluate a single condition (e.g., 'close < sma\_20')."""

try:

left, op, right = self.\_parse\_condition(condition)

left\_value = ohlcv.get(left, indicators.get(left, float(left) if left.replace('.', '').isdigit() else 0.0))

right\_value = indicators.get(right, ohlcv.get(right, float(right) if right.replace('.', '').isdigit() else 0.0))

operators = {

'>': operator.gt,

'<': operator.lt,

'=': operator.eq,

'>=': operator.ge,

'<=': operator.le,

'!=': operator.ne

}

if op not in operators:

logger.error(f"Unsupported operator in condition: {condition}")

return False

return operators[op](float(left\_value), float(right\_value))

except Exception as e:

logger.error(f"Error evaluating single condition {condition}: {e}")

return False

def \_parse\_condition(self, condition: str) -> Tuple[str, str, str]:

"""Parse a condition string into left, operator, right."""

for op in ['>=', '<=', '!=', '>', '<', '=']:

if op in condition:

left, right = condition.split(op)

return left.strip(), op, right.strip()

raise ValueError(f"Invalid condition format: {condition}")

def \_evaluate\_stop\_loss(self, symbol: str, close: float, position: Dict, indicators: Dict) -> Optional[str]:

"""Evaluate stop-loss conditions."""

try:

if not self.stop\_loss:

return None

entry\_price = position['entry\_price']

breakeven\_set = position.get('breakeven\_set', False)

if self.stop\_loss.get('type') == 'multi':

for rule in self.stop\_loss.get('rules', []):

after\_condition = rule.get('after')

if after\_condition:

if after\_condition == 'breakeven' and not breakeven\_set:

continue

if after\_condition.startswith('partial\_exit') and not position['partial\_exits'].get(after\_condition):

continue

if rule['type'] == 'fixed':

sl\_value = float(rule['value'].strip('%')) / 100

sl\_price = entry\_price \* (1 - sl\_value)

if close <= sl\_price:

logger.info(f"Fixed stop-loss triggered for {symbol}: close={close}, sl\_price={sl\_price}")

return "SELL"

elif rule['type'] == 'indicator':

indicator\_value = indicators.get(rule['indicator'], 0.0)

multiplier = rule.get('multiplier', 1.0)

sl\_price = position['highest\_price'] - (indicator\_value \* multiplier)

if close <= sl\_price:

logger.info(f"Indicator-based stop-loss triggered for {symbol}: close={close}, sl\_price={sl\_price}")

return "SELL"

elif rule['type'] == 'condition':

if self.\_evaluate\_condition(rule['condition'], {'close': close}, indicators):

logger.info(f"Condition-based stop-loss triggered for {symbol}: condition={rule['condition']}")

return "SELL"

else:

if self.stop\_loss['type'] == 'fixed':

sl\_value = float(self.stop\_loss['value'].strip('%')) / 100

sl\_price = entry\_price \* (1 - sl\_value)

if close <= sl\_price:

logger.info(f"Stop-loss triggered for {symbol}: close={close}, sl\_price={sl\_price}")

return "SELL"

elif self.stop\_loss['type'] == 'trailing':

sl\_value = float(self.stop\_loss['value'].strip('%')) / 100

position['highest\_price'] = max(position['highest\_price'], close)

sl\_price = position['highest\_price'] \* (1 - sl\_value)

if close <= sl\_price:

logger.info(f"Trailing stop-loss triggered for {symbol}: close={close}, sl\_price={sl\_price}")

return "SELL"

return None

except Exception as e:

logger.error(f"Error evaluating stop-loss for {symbol}: {e}")

return None

def \_evaluate\_target(self, symbol: str, close: float, position: Dict, indicators: Dict) -> Optional[str]:

"""Evaluate target conditions."""

try:

if not self.target:

return None

entry\_price = position['entry\_price']

breakeven\_trigger = self.trade\_management.get('breakeven', {}).get('trigger')

partial\_exits = self.trade\_management.get('partial\_exits', [])

if breakeven\_trigger and not position.get('breakeven\_set', False):

breakeven\_value = float(breakeven\_trigger) \* float(self.stop\_loss.get('rules', [{}])[0]['value'].strip('%')) / 100

breakeven\_price = entry\_price \* (1 + breakeven\_value)

if close >= breakeven\_price:

position['breakeven\_set'] = True

position['entry\_price'] = entry\_price

logger.info(f"Breakeven set for {symbol}: close={close}, breakeven\_price={breakeven\_price}")

if self.target.get('type') == 'multi':

for rule in self.target.get('rules', []):

if rule.get('partial\_exit') and not position['partial\_exits'].get(rule.get('id')):

target\_value = float(rule['value'].strip('%')) / 100

target\_price = entry\_price \* (1 + target\_value)

if close >= target\_price:

position['partial\_exits'][rule['id']] = True

logger.info(f"Partial exit triggered for {symbol} at {rule['partial\_exit']}%: close={close}, target\_price={target\_price}")

return "PARTIAL\_SELL"

elif rule['type'] == 'indicator' and rule.get('trail'):

indicator\_value = indicators.get(rule['indicator'], 0.0)

position['lowest\_price'] = min(position['lowest\_price'], close)

target\_price = indicator\_value

if close >= target\_price:

logger.info(f"Indicator-based target triggered for {symbol}: close={close}, target\_price={target\_price}")

return "SELL"

elif rule['type'] == 'condition':

if self.\_evaluate\_condition(rule['condition'], {'close': close}, indicators):

logger.info(f"Condition-based target triggered for {symbol}: condition={rule['condition']}")

return "SELL"

else:

if self.target['type'] == 'fixed':

target\_value = float(self.target['value'].strip('%')) / 100

target\_price = entry\_price \* (1 + target\_value)

if close >= target\_price:

logger.info(f"Target triggered for {symbol}: close={close}, target\_price={target\_price}")

return "SELL"

elif self.target['type'] == 'trailing':

target\_value = float(self.target['value'].strip('%')) / 100

position['lowest\_price'] = min(position['lowest\_price'], close)

target\_price = position['lowest\_price'] \* (1 + target\_value)

if close >= target\_price:

logger.info(f"Trailing target triggered for {symbol}: close={close}, target\_price={target\_price}")

return "SELL"

return None

except Exception as e:

logger.error(f"Error evaluating target for {symbol}: {e}")

return None

def \_evaluate\_scale\_in(self, symbol: str, close: float, position: Dict, indicators: Dict) -> bool:

"""Evaluate scale-in conditions."""

try:

for scale\_in in self.trade\_management.get('scale\_in', []):

if self.\_evaluate\_condition(scale\_in['condition'], {'close': close, 'entry\_price': position['entry\_price']}, indicators):

logger.info(f"Scale-in triggered for {symbol}: condition={scale\_in['condition']}, percentage={scale\_in['percentage']}%")

return True

return False

except Exception as e:

logger.error(f"Error evaluating scale-in for {symbol}: {e}")

return False

2.3.3 IndicatorEngine Update (src/rs/indicator/src/engine.rs)The Rust-based IndicatorEngine computes standard and custom indicators, storing them in Redis.

rust

*// src/rs/indicator/src/engine.rs*

use pyo3::prelude::\*;

use rdkafka::consumer::{Consumer, StreamConsumer};

use redis::AsyncCommands;

use serde\_json::Value;

use polars::prelude::\*;

#[pyclass]

pub struct IndicatorEngine {

config: Value,

consumer: StreamConsumer,

redis\_client: redis::Client,

}

#[pymethods]

impl IndicatorEngine {

#[new]

fn new(config: Value) -> Self {

let consumer = rdkafka::consumer::StreamConsumer::new(&config["kafka"]["brokers"]).unwrap();

let redis\_client = redis::Client::open(config["redis"]["host"].to\_string()).unwrap();

IndicatorEngine { config, consumer, redis\_client }

}

async fn start(&self) {

self.consumer.subscribe(&["nse\_ticks"]).unwrap();

let mut conn = self.redis\_client.get\_async\_connection().await.unwrap();

while let Some(message) = self.consumer.recv().await {

let tick: Value = serde\_json::from\_str(&message.payload().unwrap()).unwrap();

let df = self.aggregate\_ohlcv(&tick);

let indicators = self.compute\_indicators(&df);

conn.set(format!("{}:indicators:{}", tick["tradingsymbol"], self.config["timeframe"]), &indicators).await.unwrap();

}

}

fn compute\_indicators(&self, df: &DataFrame) -> Value {

let mut indicators = serde\_json::Map::new();

for ind in self.config["indicators"].as\_array().unwrap() {

match ind["type"].as\_str().unwrap() {

"rsi" => {

let rsi = df["close"].f64().unwrap().rsi(ind["period"].as\_u64().unwrap() as usize);

indicators.insert(ind["name"].to\_string(), serde\_json::to\_value(rsi.last().unwrap()).unwrap());

}

"atr" => {

let atr = df["close"].f64().unwrap().atr(df["high"].f64().unwrap(), df["low"].f64().unwrap(), ind["period"].as\_u64().unwrap() as usize);

indicators.insert(ind["name"].to\_string(), serde\_json::to\_value(atr.last().unwrap()).unwrap());

}

"sma" => {

let sma = df[ind["input"].as\_str().unwrap\_or("close")].f64().unwrap().mean(ind["period"].as\_u64().unwrap() as usize);

indicators.insert(ind["name"].to\_string(), serde\_json::to\_value(sma.last().unwrap()).unwrap());

}

"custom" => {

if ind["formula"].as\_str().unwrap().contains("vwap") {

let vwap = df["close"].f64().unwrap().mul(&df["volume"].f64().unwrap()).sum() / df["volume"].f64().unwrap().sum();

indicators.insert(ind["name"].to\_string(), serde\_json::to\_value(vwap).unwrap());

}

}

\_ => {}

}

}

serde\_json::to\_value(indicators).unwrap()

}

}

2.3.4 Configuration Update (config/config.yaml)

yaml

global:

strategies\_dir: config/markets/india/strategies

kafka:

brokers: localhost:9092

topics:

ohlcv\_1min: ohlcv\_1min

ohlcv\_5min: ohlcv\_5min

signals: signals

redis:

host: localhost

port: 6379

database:

host: localhost

port: 5432

dbname: ANQuantDB

user: anquant

password: 078692

historical\_data:

timeframes: ["1min", "5min", "30min"]

lookback\_candles: 60

markets:

india:

watchlists:

smc: config/markets/india/watchlists/smc.yaml

brokers:

angelone:

symbols: config/markets/india/brokers/symbol\_mappings/angelone.yaml

2.3.5 Testing (tests/test\_strategy\_manager.py)

python

*# tests/test\_strategy\_manager.py*

import pytest

import asyncio

import os

from src.py.core.strategy.manager import StrategyManager

from src.py.messaging.redis\_client import RedisClient

from src.py.util.config\_loader import load\_config

import yaml

import json

@pytest.mark.asyncio

async def test\_advanced\_smc\_strategy():

config = load\_config("config/config.yaml")

config["global"]["offline\_mode"] = True

redis\_client = RedisClient(config["global"]["redis"])

manager = StrategyManager(config, redis\_client)

await manager.initialize()

*# Simulate BUY signal*

ohlcv\_5min = {"close": 100.0, "high": 102.0, "low": 98.0, "volume": 1000, "tradingsymbol": "RELIANCE-EQ", "timestamp": "2025-07-18T03:00:00Z"}

ohlcv\_1min = {"close": 99.5, "high": 100.5, "low": 99.0, "volume": 900, "tradingsymbol": "RELIANCE-EQ", "timestamp": "2025-07-18T03:00:00Z"}

indicators\_5min = {"rsi": 25.0, "atr\_14": 1.0, "sma\_20": 100.0, "vwap": 99.5}

historical\_5min = [{"close": 99.0, "high": 100.0, "low": 98.0, "volume": 900, "timestamp": "2025-07-18T02:55:00Z"}] \* 20

historical\_1min = [{"close": 99.5, "high": 100.5, "low": 99.0, "volume": 900, "timestamp": "2025-07-18T02:59:00Z"}] \* 5

await redis\_client.cache("RELIANCE-EQ:ohlcv:5min", historical\_5min, ttl=86400)

await redis\_client.cache("RELIANCE-EQ:ohlcv:1min", historical\_1min, ttl=86400)

await redis\_client.cache("RELIANCE-EQ:indicators:5min", indicators\_5min, ttl=86400)

await manager.\_process\_strategy\_signals("RELIANCE-EQ", "5min", ohlcv\_5min)

signal = await redis\_client.get("signals:advanced\_smc")

assert signal, "No signal generated"

assert json.loads(signal)['signal'] == "BUY"

*# Simulate partial exit at 5% gain*

ohlcv\_5min["close"] = 105.0

await manager.\_process\_strategy\_signals("RELIANCE-EQ", "5min", ohlcv\_5min)

signal = await redis\_client.get("signals:advanced\_smc")

assert signal is None *# Partial exit, no signal*

*# Simulate condition-based exit*

indicators\_5min["rsi"] = 75.0

await redis\_client.cache("RELIANCE-EQ:indicators:5min", indicators\_5min, ttl=86400)

await manager.\_process\_strategy\_signals("RELIANCE-EQ", "5min", ohlcv\_5min)

signal = await redis\_client.get("signals:advanced\_smc")

assert json.loads(signal)['signal'] == "SELL"

3. Integration with ANQuant Architecture

* Pre-Fetch (4:00 PM IST, July 17, 2025):
  + scripts/prefetch\_historical\_data.py fetches 60 candles for stocks in config/markets/india/watchlists/smc.yaml, writes to Redis (symbol:ohlcv:5min, symbol:ohlcv:1min) and PostgreSQL (ohlcv table).
* Startup (9:30 AM IST, July 18, 2025):
  + src/py/core/historical\_data\_manager.py loads OHLCV, adjusts for corporate actions (e.g., SUNPHARMA dividend ₹5.50), and updates Redis/PostgreSQL.
* Real-Time Trading:
  + src/rs/market\_data/src/engine.rs streams ticks to Kafka nse\_ticks.
  + src/rs/indicator/src/engine.rs computes indicators, writes to Redis (symbol:indicators:5min, symbol:indicators:1min) and Kafka (ohlcv\_5min, ohlcv\_1min).
  + src/py/core/strategy/manager.py evaluates rules, publishes signals to Kafka (signals) and Redis (signals:advanced\_smc).

4. Effort Estimate

* StrategyManager Implementation: 8–12 hours.
* RuleEngine Implementation: 12–18 hours.
* IndicatorEngine Update: 4–6 hours.
* YAML Creation: 4–6 hours.
* Testing: 8–12 hours.
* Total: ~36–54 hours.
* Timeline: Complete by early August 2025.

5. Recommendations

1. Implement FlexiRule:
   * Create src/py/core/strategy/manager.py and rules.py.
   * Populate config/markets/india/strategies/advanced\_smc.yaml.
2. Enhance Security:
   * Use asteval for safe rule evaluation, avoiding eval.
3. Test Thoroughly:
   * Run tests/test\_strategy\_manager.py with a small watchlist (10 stocks).
   * Verify logs in logs/strategy/advanced\_smc\_2025-07-18.log.
4. Future Improvements:
   * Add support for dynamic risk-reward ratios and advanced position scaling.
   * Optimize Redis caching for large watchlists.

6. ConclusionThe FlexiRule framework meets all functional and non-functional requirements, supporting flexible strategy definitions, multi-timeframe analysis, and advanced stop-loss/target mechanisms. The Advanced SMC Strategy example demonstrates its capabilities, covering market structure, order blocks, liquidity pools, confirmation, and trade management. The implementation integrates seamlessly with the ANQuant architecture, ensuring scalability and performance. Implement and test by early August 2025 to enable robust trading. For future references, store this documentation in docs/flexirule\_design.md and let me know if you need additional details or assistance!