from \_\_future\_\_ import annotations  
  
# =============== Standard Library Imports ===============  
import os  
import sys  
import gc  
import csv  
import ssl  
import math  
import json  
import time  
import uuid  
import glob  
import queue  
import errno  
import atexit  
import base64  
import random  
import sqlite3  
import logging  
import threading  
import traceback  
import functools  
import itertools  
import statistics as stats  
import weakref  
from dataclasses import dataclass, field, asdict  
from typing import (  
 Any, Dict, List, Optional, Tuple, Callable, Union, Protocol,   
 TypeVar, Generic, Iterator, AsyncIterator, NamedTuple, Literal  
)  
from datetime import datetime, timedelta, timezone  
from pathlib import Path  
from contextlib import asynccontextmanager, contextmanager  
from concurrent.futures import ThreadPoolExecutor  
from collections import defaultdict, deque  
import asyncio  
from asyncio import Semaphore, Lock as AsyncLock  
  
# =============== Custom Exception Classes ===============  
class LokiException(Exception):  
 """Base exception for Loki trading system"""  
 pass  
  
class ConfigurationError(LokiException):  
 """Configuration validation errors"""  
 pass  
  
class ExchangeError(LokiException):  
 """Exchange connection and API errors"""  
 pass  
  
class RiskManagementError(LokiException):  
 """Risk management constraint violations"""  
 pass  
  
class ModelError(LokiException):  
 """Model loading and inference errors"""  
 pass  
  
class DataValidationError(LokiException):  
 """Data validation and integrity errors"""  
 pass  
  
# =============== Optional Third-Party Imports (Graceful Degrade) ===============  
try:  
 import numpy as np  
 HAS\_NUMPY = True  
except ImportError:  
 np = None  
 HAS\_NUMPY = False  
  
try:  
 import pandas as pd  
 HAS\_PANDAS = True  
except ImportError:  
 pd = None  
 HAS\_PANDAS = False  
  
try:  
 import ccxt.async\_support as ccxt\_async  
 HAS\_CCXT = True  
except ImportError:  
 ccxt\_async = None  
 HAS\_CCXT = False  
  
try:  
 from cryptography.fernet import Fernet  
 HAS\_CRYPTO = True  
except ImportError:  
 Fernet = None  
 HAS\_CRYPTO = False  
  
# =============== Type Aliases ===============  
T = TypeVar('T')  
ConfigDict = Dict[str, Any]  
PositionDict = Dict[str, Dict[str, float]]  
PriceDict = Dict[str, float]  
ModelDict = Dict[str, Any]  
  
# =============== Protocol Definitions ===============  
class ModelProtocol(Protocol):  
 def predict(self, features: np.ndarray) -> Union[np.ndarray, List[float]]:  
 ...  
  
class ExchangeProtocol(Protocol):  
 async def fetch\_ohlcv(self, symbol: str, timeframe: str, limit: int) -> Optional['pd.DataFrame']:  
 ...  
   
 async def get\_ticker(self, symbol: str) -> Optional[Dict[str, Any]]:  
 ...  
  
# =============== Constants & Paths ===============  
APP\_DIR = Path(\_\_file\_\_).resolve().parent  
DATA\_DIR = APP\_DIR / "data"  
MODELS\_DIR = APP\_DIR / "models"  
LOGS\_DIR = APP\_DIR / "logs"  
REPORTS\_DIR = APP\_DIR / "reports"  
DB\_PATH = DATA\_DIR / "loki.sqlite3"  
MODELS\_REGISTRY = APP\_DIR / "models.json"  
FERNET\_KEY\_FILE = DATA\_DIR / ".fernet.key"  
CONFIG\_FILE = APP\_DIR / "config.json"  
  
# Create directories  
for directory in (DATA\_DIR, MODELS\_DIR, LOGS\_DIR, REPORTS\_DIR):  
 directory.mkdir(parents=True, exist\_ok=True)  
  
# =============== Enhanced Logging with Correlation IDs ===============  
class CorrelationIDFilter(logging.Filter):  
 def filter(self, record: logging.LogRecord) -> bool:  
 if not hasattr(record, 'correlation\_id'):  
 record.correlation\_id = getattr(threading.current\_thread(), 'correlation\_id', 'MAIN')  
 return True  
  
def setup\_logging() -> logging.Logger:  
 """Setup structured logging with correlation IDs"""  
 log\_level = os.getenv("LOKI\_LOG\_LEVEL", "INFO").upper()  
   
 logger = logging.getLogger("loki")  
 logger.setLevel(log\_level)  
   
 # Clear existing handlers  
 logger.handlers.clear()  
   
 # Console handler  
 console\_handler = logging.StreamHandler(sys.stdout)  
 console\_handler.setLevel(log\_level)  
   
 # File handler with rotation  
 file\_handler = logging.FileHandler(LOGS\_DIR / "loki.log", encoding="utf-8")  
 file\_handler.setLevel(log\_level)  
   
 # Formatter with correlation ID  
 formatter = logging.Formatter(  
 fmt="%(asctime)s | %(levelname)s | %(correlation\_id)s | %(name)s | %(message)s",  
 datefmt="%Y-%m-%d %H:%M:%S",  
 )  
   
 # Add correlation ID filter  
 correlation\_filter = CorrelationIDFilter()  
   
 for handler in [console\_handler, file\_handler]:  
 handler.setFormatter(formatter)  
 handler.addFilter(correlation\_filter)  
 logger.addHandler(handler)  
   
 return logger  
  
logger = setup\_logging()  
  
def set\_correlation\_id(correlation\_id: str) -> None:  
 """Set correlation ID for current thread"""  
 threading.current\_thread().correlation\_id = correlation\_id  
  
def log\_with\_context(level: str, msg: str, \*\*ctx: Any) -> None:  
 """Log with structured context"""  
 log\_func = getattr(logger, level.lower(), logger.info)  
 if ctx:  
 log\_func("%s | ctx=%s", msg, json.dumps(ctx, default=str))  
 else:  
 log\_func(msg)  
  
def log\_exception(msg: str, exc: Optional[Exception] = None, \*\*ctx: Any) -> None:  
 """Log exception with context"""  
 if exc:  
 ctx['exception\_type'] = type(exc).\_\_name\_\_  
 ctx['exception\_msg'] = str(exc)  
   
 logger.error("%s | ctx=%s\n%s", msg, json.dumps(ctx, default=str), traceback.format\_exc())  
  
# =============== Utility Functions ===============  
def utcnow() -> datetime:  
 """Get current UTC datetime"""  
 return datetime.now(timezone.utc)  
  
def utc\_timestamp() -> float:  
 """Get current UTC timestamp"""  
 return utcnow().timestamp()  
  
def safe\_float(value: Any, default: float = 0.0) -> float:  
 """Safely convert value to float"""  
 try:  
 return float(value)  
 except (TypeError, ValueError):  
 return default  
  
def safe\_int(value: Any, default: int = 0) -> int:  
 """Safely convert value to int"""  
 try:  
 return int(value)  
 except (TypeError, ValueError):  
 return default  
  
def clamp(value: float, min\_val: float, max\_val: float) -> float:  
 """Clamp value between min and max"""  
 return max(min\_val, min(max\_val, value))  
  
def human\_datetime(timestamp: Optional[float] = None) -> str:  
 """Convert timestamp to human readable datetime"""  
 ts = timestamp or time.time()  
 return datetime.fromtimestamp(ts).strftime("%Y-%m-%d %H:%M:%S")  
  
# =============== Enhanced Circuit Breaker ===============  
@dataclass  
class CircuitBreakerState:  
 """Circuit breaker state tracking"""  
 fail\_count: int = 0  
 state: Literal["CLOSED", "OPEN", "HALF\_OPEN"] = "CLOSED"  
 last\_failure\_time: float = 0.0  
 last\_success\_time: float = 0.0  
  
class CircuitBreaker:  
 """Enhanced circuit breaker with exponential backoff"""  
   
 def \_\_init\_\_(  
 self,   
 failure\_threshold: int = 5,   
 reset\_timeout: float = 30.0,  
 half\_open\_max\_calls: int = 3  
 ):  
 self.failure\_threshold = failure\_threshold  
 self.reset\_timeout = reset\_timeout  
 self.half\_open\_max\_calls = half\_open\_max\_calls  
 self.\_state = CircuitBreakerState()  
 self.\_lock = threading.Lock()  
 self.\_half\_open\_calls = 0  
   
 def can\_execute(self) -> bool:  
 """Check if execution is allowed"""  
 with self.\_lock:  
 current\_time = time.time()  
   
 if self.\_state.state == "CLOSED":  
 return True  
 elif self.\_state.state == "OPEN":  
 if current\_time - self.\_state.last\_failure\_time >= self.reset\_timeout:  
 self.\_state.state = "HALF\_OPEN"  
 self.\_half\_open\_calls = 0  
 return True  
 return False  
 elif self.\_state.state == "HALF\_OPEN":  
 return self.\_half\_open\_calls < self.half\_open\_max\_calls  
   
 return False  
   
 def record\_success(self) -> None:  
 """Record successful execution"""  
 with self.\_lock:  
 self.\_state.fail\_count = 0  
 self.\_state.last\_success\_time = time.time()  
 if self.\_state.state == "HALF\_OPEN":  
 self.\_state.state = "CLOSED"  
 self.\_half\_open\_calls = 0  
 elif self.\_state.state == "OPEN":  
 self.\_state.state = "CLOSED"  
   
 def record\_failure(self) -> None:  
 """Record failed execution"""  
 with self.\_lock:  
 self.\_state.fail\_count += 1  
 self.\_state.last\_failure\_time = time.time()  
   
 if self.\_state.state == "HALF\_OPEN":  
 self.\_half\_open\_calls += 1  
 if self.\_half\_open\_calls >= self.half\_open\_max\_calls:  
 self.\_state.state = "OPEN"  
 elif self.\_state.fail\_count >= self.failure\_threshold:  
 self.\_state.state = "OPEN"  
   
 @property  
 def state(self) -> str:  
 """Get current circuit breaker state"""  
 return self.\_state.state  
  
# =============== Enhanced Retry Decorator ===============  
def retry\_async(  
 max\_attempts: int = 3,  
 base\_delay: float = 0.5,  
 max\_delay: float = 30.0,  
 exponential\_base: float = 2.0,  
 jitter: bool = True,  
 exceptions: Tuple[type, ...] = (Exception,)  
):  
 """Enhanced async retry decorator with exponential backoff"""  
 def decorator(func: Callable) -> Callable:  
 @functools.wraps(func)  
 async def wrapper(\*args, \*\*kwargs):  
 last\_exception = None  
   
 for attempt in range(max\_attempts):  
 try:  
 return await func(\*args, \*\*kwargs)  
 except exceptions as e:  
 last\_exception = e  
   
 if attempt == max\_attempts - 1:  
 break  
   
 # Calculate delay with exponential backoff  
 delay = min(base\_delay \* (exponential\_base \*\* attempt), max\_delay)  
   
 # Add jitter to prevent thundering herd  
 if jitter:  
 delay \*= (0.5 + random.random() \* 0.5)  
   
 log\_with\_context(  
 "warning",   
 f"Attempt {attempt + 1} failed, retrying in {delay:.2f}s",  
 function=func.\_\_name\_\_,  
 exception=str(e)  
 )  
   
 await asyncio.sleep(delay)  
   
 raise last\_exception  
   
 return wrapper  
 return decorator  
  
# =============== Memory-Efficient Data Manager ===============  
class MemoryEfficientDataFrame:  
 """Memory-efficient DataFrame wrapper with automatic cleanup"""  
   
 def \_\_init\_\_(self, df: 'pd.DataFrame', max\_size: int = 1000):  
 self.max\_size = max\_size  
 self.\_df = df.tail(max\_size) if len(df) > max\_size else df.copy()  
 self.\_last\_accessed = time.time()  
   
 @property  
 def df(self) -> 'pd.DataFrame':  
 """Get DataFrame with access tracking"""  
 self.\_last\_accessed = time.time()  
 return self.\_df  
   
 def append\_row(self, row: Dict[str, Any]) -> None:  
 """Append row with automatic size management"""  
 if not HAS\_PANDAS:  
 return  
   
 new\_row = pd.DataFrame([row], index=[row.get('timestamp', pd.Timestamp.now())])  
 self.\_df = pd.concat([self.\_df, new\_row]).tail(self.max\_size)  
 self.\_last\_accessed = time.time()  
   
 def is\_stale(self, max\_age\_seconds: float = 3600) -> bool:  
 """Check if data is stale"""  
 return time.time() - self.\_last\_accessed > max\_age\_seconds  
   
 def memory\_usage(self) -> int:  
 """Get memory usage in bytes"""  
 if not HAS\_PANDAS:  
 return 0  
 return self.\_df.memory\_usage(deep=True).sum()  
  
class DataManager:  
 """Thread-safe data manager with memory optimization"""  
   
 def \_\_init\_\_(self, max\_symbols: int = 100, max\_rows\_per\_symbol: int = 1000):  
 self.max\_symbols = max\_symbols  
 self.max\_rows\_per\_symbol = max\_rows\_per\_symbol  
 self.\_data: Dict[str, MemoryEfficientDataFrame] = {}  
 self.\_lock = threading.RLock()  
 self.\_cleanup\_interval = 3600 # 1 hour  
 self.\_last\_cleanup = time.time()  
   
 def store\_ohlcv(self, symbol: str, timeframe: str, df: 'pd.DataFrame') -> None:  
 """Store OHLCV data with memory management"""  
 if not HAS\_PANDAS or df is None or df.empty:  
 return  
   
 key = f"{symbol}\_{timeframe}"  
   
 with self.\_lock:  
 # Cleanup if needed  
 self.\_cleanup\_if\_needed()  
   
 # Manage symbol limit  
 if key not in self.\_data and len(self.\_data) >= self.max\_symbols:  
 # Remove oldest accessed data  
 oldest\_key = min(  
 self.\_data.keys(),   
 key=lambda k: self.\_data[k].\_last\_accessed  
 )  
 del self.\_data[oldest\_key]  
 gc.collect()  
   
 # Store or update data  
 if key in self.\_data:  
 # Append new data if it's newer  
 existing\_df = self.\_data[key].df  
 if not existing\_df.empty:  
 last\_timestamp = existing\_df.index[-1]  
 new\_data = df[df.index > last\_timestamp]  
 if not new\_data.empty:  
 for \_, row in new\_data.iterrows():  
 self.\_data[key].append\_row(row.to\_dict())  
 else:  
 self.\_data[key] = MemoryEfficientDataFrame(df, self.max\_rows\_per\_symbol)  
 else:  
 self.\_data[key] = MemoryEfficientDataFrame(df, self.max\_rows\_per\_symbol)  
   
 def get\_ohlcv(self, symbol: str, timeframe: str) -> Optional['pd.DataFrame']:  
 """Get OHLCV data"""  
 key = f"{symbol}\_{timeframe}"  
   
 with self.\_lock:  
 if key in self.\_data:  
 return self.\_data[key].df.copy()  
 return None  
   
 def \_cleanup\_if\_needed(self) -> None:  
 """Cleanup stale data if needed"""  
 current\_time = time.time()  
 if current\_time - self.\_last\_cleanup < self.\_cleanup\_interval:  
 return  
   
 stale\_keys = [  
 key for key, data in self.\_data.items()   
 if data.is\_stale()  
 ]  
   
 for key in stale\_keys:  
 del self.\_data[key]  
   
 if stale\_keys:  
 gc.collect()  
 log\_with\_context("info", f"Cleaned up {len(stale\_keys)} stale datasets")  
   
 self.\_last\_cleanup = current\_time  
   
 def get\_memory\_usage(self) -> Dict[str, int]:  
 """Get memory usage statistics"""  
 with self.\_lock:  
 return {  
 key: data.memory\_usage()   
 for key, data in self.\_data.items()  
 }  
   
 def clear(self) -> None:  
 """Clear all data"""  
 with self.\_lock:  
 self.\_data.clear()  
 gc.collect()  
  
# =============== Enhanced Configuration Management ===============  
DEFAULT\_CONFIG: ConfigDict = {  
 "paper": True,  
 "exchange": "binance",  
 "assets": ["BTC/USDT"],  
 "timeframes": ["1h", "4h", "1d"],  
 "poll\_interval": 5.0,  
 "risk": {  
 "max\_drawdown\_pct": 0.50,  
 "daily\_loss\_limit\_pct": 0.10,  
 "profit\_anchor\_pct": 0.10,  
 "stop\_loss\_pct": 0.02,  
 "trailing\_stop\_pct": 0.015,  
 "max\_single\_pos\_pct": 0.15,  
 "max\_portfolio\_exposure\_pct": 0.80,  
 "var\_window": 200,  
 "var\_quantile": 0.99  
 },  
 "positioning": {  
 "atr\_period": 14,  
 "atr\_risk\_mult": 1.0,  
 "vol\_regimes": [0.5, 1.5],  
 "correlation\_limit": 0.85  
 },  
 "signals": {  
 "rsi\_period": 14,  
 "rsi\_buy": 30,  
 "rsi\_sell": 70,  
 "bb\_period": 20,  
 "bb\_mult": 2.0,  
 "ema\_smooth": 5,  
 "adx\_period": 14,  
 "stoch\_k": 14,  
 "stoch\_d": 3  
 },  
 "weights": {  
 "ta": 0.6,  
 "ml": 0.3,  
 "llm": 0.1  
 },  
 "alerts": {  
 "email": False,  
 "slack": False  
 },  
 "performance": {  
 "max\_concurrent\_requests": 8,  
 "request\_timeout": 30.0,  
 "max\_memory\_mb": 1024,  
 "cleanup\_interval": 3600  
 }  
}  
  
class ConfigValidator:  
 """Enhanced configuration validator"""  
   
 @staticmethod  
 def validate\_numeric\_range(  
 value: Any,   
 min\_val: float,   
 max\_val: float,   
 field\_name: str  
 ) -> float:  
 """Validate numeric value is within range"""  
 try:  
 num\_val = float(value)  
 if not (min\_val <= num\_val <= max\_val):  
 raise ConfigurationError(  
 f"{field\_name} must be between {min\_val} and {max\_val}, got {num\_val}"  
 )  
 return num\_val  
 except (TypeError, ValueError) as e:  
 raise ConfigurationError(f"{field\_name} must be a valid number: {e}")  
   
 @staticmethod  
 def validate\_config(config: ConfigDict) -> None:  
 """Comprehensive configuration validation"""  
 # Validate signal parameters  
 signals = config.get("signals", {})  
   
 ConfigValidator.validate\_numeric\_range(  
 signals.get("rsi\_period", 14), 1, 100, "signals.rsi\_period"  
 )  
 ConfigValidator.validate\_numeric\_range(  
 signals.get("bb\_period", 20), 1, 300, "signals.bb\_period"  
 )  
   
 rsi\_buy = ConfigValidator.validate\_numeric\_range(  
 signals.get("rsi\_buy", 30), 0, 100, "signals.rsi\_buy"  
 )  
 rsi\_sell = ConfigValidator.validate\_numeric\_range(  
 signals.get("rsi\_sell", 70), 0, 100, "signals.rsi\_sell"  
 )  
   
 if rsi\_buy >= rsi\_sell:  
 raise ConfigurationError("signals.rsi\_buy must be less than signals.rsi\_sell")  
   
 # Validate risk parameters  
 risk = config.get("risk", {})  
 for key in ["max\_drawdown\_pct", "daily\_loss\_limit\_pct", "profit\_anchor\_pct",  
 "stop\_loss\_pct", "trailing\_stop\_pct", "max\_single\_pos\_pct",   
 "max\_portfolio\_exposure\_pct"]:  
 ConfigValidator.validate\_numeric\_range(  
 risk.get(key, 0.0), 0.0, 1.0, f"risk.{key}"  
 )  
   
 # Validate positioning parameters  
 positioning = config.get("positioning", {})  
 ConfigValidator.validate\_numeric\_range(  
 positioning.get("correlation\_limit", 0.85), -1.0, 1.0,   
 "positioning.correlation\_limit"  
 )  
   
 # Validate assets format  
 assets = config.get("assets", [])  
 if not assets:  
 raise ConfigurationError("At least one asset must be specified")  
   
 for asset in assets:  
 if not isinstance(asset, str) or "/" not in asset:  
 raise ConfigurationError(f"Invalid asset format: {asset}. Expected format: 'BASE/QUOTE'")  
   
 # Validate poll interval  
 ConfigValidator.validate\_numeric\_range(  
 config.get("poll\_interval", 5.0), 1.0, 3600.0, "poll\_interval"  
 )  
   
 # Validate performance parameters  
 performance = config.get("performance", {})  
 ConfigValidator.validate\_numeric\_range(  
 performance.get("max\_concurrent\_requests", 8), 1, 128,   
 "performance.max\_concurrent\_requests"  
 )  
  
def load\_config() -> ConfigDict:  
 """Load and validate configuration"""  
 try:  
 if CONFIG\_FILE.exists():  
 with open(CONFIG\_FILE, 'r', encoding='utf-8') as f:  
 config = json.load(f)  
   
 # Merge with defaults for missing keys  
 def merge\_configs(default: Dict, user: Dict) -> Dict:  
 result = default.copy()  
 for key, value in user.items():  
 if key in result and isinstance(result[key], dict) and isinstance(value, dict):  
 result[key] = merge\_configs(result[key], value)  
 else:  
 result[key] = value  
 return result  
   
 config = merge\_configs(DEFAULT\_CONFIG, config)  
 else:  
 config = DEFAULT\_CONFIG.copy()  
 save\_config(config)  
 log\_with\_context("info", "Created default configuration file")  
   
 ConfigValidator.validate\_config(config)  
 return config  
   
 except json.JSONDecodeError as e:  
 raise ConfigurationError(f"Invalid JSON in config file: {e}")  
 except Exception as e:  
 log\_exception("Failed to load configuration", e)  
 return DEFAULT\_CONFIG.copy()  
  
def save\_config(config: ConfigDict) -> None:  
 """Save configuration to file"""  
 try:  
 with open(CONFIG\_FILE, 'w', encoding='utf-8') as f:  
 json.dump(config, f, indent=2)  
 except Exception as e:  
 log\_exception("Failed to save configuration", e)  
  
# =============== Enhanced Database Layer ===============  
class DatabaseManager:  
 """Thread-safe database manager with connection pooling"""  
   
 def \_\_init\_\_(self, db\_path: Path, max\_connections: int = 5):  
 self.db\_path = db\_path  
 self.max\_connections = max\_connections  
 self.\_connections: queue.Queue = queue.Queue(maxsize=max\_connections)  
 self.\_lock = threading.Lock()  
 self.\_initialized = False  
   
 def \_create\_connection(self) -> sqlite3.Connection:  
 """Create a new database connection"""  
 conn = sqlite3.connect(  
 str(self.db\_path),   
 check\_same\_thread=False,  
 timeout=30.0  
 )  
 conn.execute("PRAGMA journal\_mode=WAL;")  
 conn.execute("PRAGMA synchronous=NORMAL;")  
 conn.execute("PRAGMA foreign\_keys=ON;")  
 conn.row\_factory = sqlite3.Row  
 return conn  
   
 def initialize(self) -> None:  
 """Initialize database schema and connection pool"""  
 if self.\_initialized:  
 return  
   
 with self.\_lock:  
 if self.\_initialized:  
 return  
   
 # Create initial connections  
 for \_ in range(self.max\_connections):  
 conn = self.\_create\_connection()  
 self.\_connections.put(conn)  
   
 # Initialize schema  
 with self.get\_connection() as conn:  
 self.\_create\_schema(conn)  
   
 self.\_initialized = True  
   
 @contextmanager  
 def get\_connection(self):  
 """Get database connection from pool"""  
 if not self.\_initialized:  
 self.initialize()  
   
 try:  
 conn = self.\_connections.get(timeout=10.0)  
 yield conn  
 except queue.Empty:  
 raise ExchangeError("Database connection timeout")  
 finally:  
 try:  
 self.\_connections.put(conn, timeout=1.0)  
 except queue.Full:  
 conn.close()  
   
 def \_create\_schema(self, conn: sqlite3.Connection) -> None:  
 """Create database schema"""  
 schemas = {  
 "trades": """  
 CREATE TABLE IF NOT EXISTS trades(  
 id TEXT PRIMARY KEY,  
 ts REAL NOT NULL,  
 symbol TEXT NOT NULL,  
 side TEXT NOT NULL,  
 qty REAL NOT NULL,  
 price REAL NOT NULL,  
 fee REAL DEFAULT 0.0,  
 pnl REAL DEFAULT 0.0,  
 strategy TEXT DEFAULT 'core',  
 notes TEXT,  
 correlation\_id TEXT,  
 created\_at REAL DEFAULT (unixepoch())  
 );  
 """,  
 "orders": """  
 CREATE TABLE IF NOT EXISTS orders(  
 id TEXT PRIMARY KEY,  
 ts REAL NOT NULL,  
 symbol TEXT NOT NULL,  
 side TEXT NOT NULL,  
 type TEXT NOT NULL,  
 qty REAL NOT NULL,  
 price REAL NOT NULL,  
 status TEXT NOT NULL,  
 parent\_id TEXT,  
 expires\_ts REAL,  
 correlation\_id TEXT,  
 created\_at REAL DEFAULT (unixepoch())  
 );  
 """,  
 "positions": """  
 CREATE TABLE IF NOT EXISTS positions(  
 symbol TEXT PRIMARY KEY,  
 qty REAL NOT NULL DEFAULT 0.0,  
 avg\_price REAL NOT NULL DEFAULT 0.0,  
 sector TEXT,  
 region TEXT,  
 updated\_at REAL DEFAULT (unixepoch())  
 );  
 """,  
 "metrics": """  
 CREATE TABLE IF NOT EXISTS metrics(  
 ts REAL NOT NULL,  
 name TEXT NOT NULL,  
 value REAL NOT NULL,  
 metadata TEXT,  
 correlation\_id TEXT,  
 PRIMARY KEY(ts, name)  
 );  
 """,  
 "audit": """  
 CREATE TABLE IF NOT EXISTS audit(  
 ts REAL NOT NULL DEFAULT (unixepoch()),  
 level TEXT NOT NULL,  
 message TEXT NOT NULL,  
 context TEXT,  
 correlation\_id TEXT  
 );  
 """  
 }  
   
 cursor = conn.cursor()  
 for table\_name, schema in schemas.items():  
 try:  
 cursor.execute(schema)  
 # Create indexes  
 if table\_name == "trades":  
 cursor.execute("CREATE INDEX IF NOT EXISTS idx\_trades\_symbol\_ts ON trades(symbol, ts);")  
 cursor.execute("CREATE INDEX IF NOT EXISTS idx\_trades\_correlation ON trades(correlation\_id);")  
 elif table\_name == "metrics":  
 cursor.execute("CREATE INDEX IF NOT EXISTS idx\_metrics\_name\_ts ON metrics(name, ts);")  
 except sqlite3.Error as e:  
 log\_exception(f"Failed to create {table\_name} schema", e)  
   
 conn.commit()  
   
 def insert\_record(self, table: str, record: Dict[str, Any]) -> bool:  
 """Insert record with automatic correlation ID"""  
 try:  
 # Add correlation ID if not present  
 if 'correlation\_id' not in record:  
 record['correlation\_id'] = getattr(threading.current\_thread(), 'correlation\_id', 'UNKNOWN')  
   
 with self.get\_connection() as conn:  
 columns = list(record.keys())  
 placeholders = ', '.join(['?' for \_ in columns])  
 values = [record[col] for col in columns]  
   
 query = f"INSERT OR REPLACE INTO {table} ({', '.join(columns)}) VALUES ({placeholders})"  
 conn.execute(query, values)  
 conn.commit()  
 return True  
   
 except sqlite3.Error as e:  
 log\_exception(f"Failed to insert into {table}", e, record=record)  
 return False  
   
 def query\_records(  
 self,   
 query: str,   
 params: Tuple = (),   
 fetch\_size: Optional[int] = None  
 ) -> List[Dict[str, Any]]:  
 """Query records with optional result limiting"""  
 try:  
 with self.get\_connection() as conn:  
 cursor = conn.execute(query, params)  
   
 if fetch\_size:  
 rows = cursor.fetchmany(fetch\_size)  
 else:  
 rows = cursor.fetchall()  
   
 return [dict(row) for row in rows]  
   
 except sqlite3.Error as e:  
 log\_exception("Database query failed", e, query=query, params=params)  
 return []  
   
 def close(self) -> None:  
 """Close all connections"""  
 while not self.\_connections.empty():  
 try:  
 conn = self.\_connections.get\_nowait()  
 conn.close()  
 except queue.Empty:  
 break  
  
# =============== Enhanced Exchange Manager ===============  
class ExchangeManager:  
 """Enhanced exchange manager with better error handling and caching"""  
   
 def \_\_init\_\_(self, config: ConfigDict, semaphore: Semaphore, data\_manager: DataManager):  
 self.config = config  
 self.paper\_mode = bool(config.get("paper", True)) or not HAS\_CCXT  
 self.exchange\_name = str(config.get("exchange", "binance")).lower()  
 self.semaphore = semaphore  
 self.data\_manager = data\_manager  
   
 # Circuit breakers for different operations  
 self.fetch\_cb = CircuitBreaker(failure\_threshold=5, reset\_timeout=60.0)  
 self.ticker\_cb = CircuitBreaker(failure\_threshold=3, reset\_timeout=30.0)  
   
 # Exchange instance  
 self.\_exchange: Optional[Any] = None  
 self.\_exchange\_lock = AsyncLock()  
   
 # Performance tracking  
 self.\_request\_times: deque = deque(maxlen=100)  
 self.\_last\_rate\_limit\_reset = time.time()  
 self.\_rate\_limit\_remaining = 1000  
   
 async def initialize(self) -> None:  
 """Initialize exchange connection"""  
 if self.paper\_mode:  
 log\_with\_context("info", "Exchange manager in PAPER mode")  
 return  
   
 if not HAS\_CCXT:  
 log\_with\_context("warning", "CCXT not available, falling back to paper mode")  
 self.paper\_mode = True  
 return  
   
 try:  
 async with self.\_exchange\_lock:  
 if self.\_exchange is not None:  
 return  
   
 exchange\_class = getattr(ccxt\_async, self.exchange\_name, None)  
 if not exchange\_class:  
 raise ExchangeError(f"Exchange {self.exchange\_name} not supported")  
   
 # Load credentials if available  
 credentials = self.\_load\_credentials()  
   
 self.\_exchange = exchange\_class({  
 "enableRateLimit": True,  
 "timeout": self.config.get("performance", {}).get("request\_timeout", 30.0) \* 1000,  
 \*\*credentials  
 })  
   
 # Test connection  
 await self.\_exchange.load\_markets()  
 log\_with\_context("info", f"Exchange {self.exchange\_name} initialized successfully")  
   
 except Exception as e:  
 log\_exception("Exchange initialization failed, falling back to paper mode", e)  
 self.paper\_mode = True  
 self.\_exchange = None  
   
 def \_load\_credentials(self) -> Dict[str, str]:  
 """Load encrypted exchange credentials"""  
 if not HAS\_CRYPTO:  
 return {}  
   
 cred\_file = DATA\_DIR / f"{self.exchange\_name}.enc"  
 if not cred\_file.exists():  
 return {}  
   
 try:  
 return self.\_decrypt\_credentials(cred\_file)  
 except Exception as e:  
 log\_exception("Failed to load credentials", e)  
 return {}  
   
 def \_decrypt\_credentials(self, file\_path: Path) -> Dict[str, str]:  
 """Decrypt stored credentials"""  
 if not HAS\_CRYPTO or not FERNET\_KEY\_FILE.exists():  
 return {}  
   
 try:  
 key = FERNET\_KEY\_FILE.read\_bytes()  
 fernet = Fernet(key)  
   
 encrypted\_data = file\_path.read\_bytes()  
 decrypted\_data = fernet.decrypt(encrypted\_data)  
   
 return json.loads(decrypted\_data.decode('utf-8'))  
 except Exception as e:  
 raise ExchangeError(f"Failed to decrypt credentials: {e}")  
   
 @retry\_async(max\_attempts=3, base\_delay=1.0, exceptions=(ExchangeError, asyncio.TimeoutError))  
 async def fetch\_ohlcv(  
 self,   
 symbol: str,   
 timeframe: str = "1h",   
 limit: int = 256  
 ) -> Optional['pd.DataFrame']:  
 """Fetch OHLCV data with caching and error handling"""  
 if not HAS\_PANDAS:  
 log\_with\_context("warning", "Pandas not available for OHLCV data")  
 return None  
   
 # Check circuit breaker  
 if not self.fetch\_cb.can\_execute():  
 log\_with\_context("warning", "OHLCV fetch blocked by circuit breaker")  
 await asyncio.sleep(1.0)  
 return None  
   
 try:  
 # Check cache first  
 cached\_data = self.data\_manager.get\_ohlcv(symbol, timeframe)  
 if cached\_data is not None and len(cached\_data) >= limit \* 0.8:  
 self.fetch\_cb.record\_success()  
 return cached\_data.tail(limit)  
   
 async with self.semaphore:  
 start\_time = time.time()  
   
 if self.paper\_mode or self.\_exchange is None:  
 df = self.\_generate\_synthetic\_ohlcv(symbol, timeframe, limit)  
 else:  
 raw\_data = await self.\_exchange.fetch\_ohlcv(  
 symbol, timeframe=timeframe, limit=limit  
 )  
 df = self.\_process\_raw\_ohlcv(raw\_data, symbol)  
   
 # Track performance  
 request\_time = time.time() - start\_time  
 self.\_request\_times.append(request\_time)  
   
 if df is not None and not df.empty:  
 # Cache the data  
 self.data\_manager.store\_ohlcv(symbol, timeframe, df)  
 self.fetch\_cb.record\_success()  
   
 log\_with\_context(  
 "debug",   
 f"Fetched {len(df)} OHLCV records",  
 symbol=symbol,   
 timeframe=timeframe,  
 request\_time=f"{request\_time:.3f}s"  
 )  
   
 return df  
   
 except Exception as e:  
 self.fetch\_cb.record\_failure()  
 log\_exception("OHLCV fetch failed", e, symbol=symbol, timeframe=timeframe)  
 return None  
   
 def \_generate\_synthetic\_ohlcv(  
 self,   
 symbol: str,   
 timeframe: str,   
 limit: int  
 ) -> Optional['pd.DataFrame']:  
 """Generate synthetic OHLCV data for paper trading"""  
 if not HAS\_PANDAS or not HAS\_NUMPY:  
 return None  
   
 try:  
 # Create realistic price movement  
 end\_time = pd.Timestamp.utcnow()  
 freq\_map = {"1m": "1T", "5m": "5T", "15m": "15T", "1h": "1H", "4h": "4H", "1d": "1D"}  
 freq = freq\_map.get(timeframe, "1H")  
   
 index = pd.date\_range(end=end\_time, periods=limit, freq=freq)  
   
 # Generate realistic price data with volatility  
 base\_price = 30000.0 + np.random.normal(0, 1000, 1)[0]  
 returns = np.random.normal(0, 0.002, limit) # 0.2% daily volatility  
   
 # Apply some trending behavior  
 trend = np.linspace(-0.001, 0.001, limit)  
 returns += trend  
   
 prices = base\_price \* np.exp(np.cumsum(returns))  
   
 # Generate OHLC from closes  
 highs = prices \* (1 + np.abs(np.random.normal(0, 0.001, limit)))  
 lows = prices \* (1 - np.abs(np.random.normal(0, 0.001, limit)))  
 opens = np.roll(prices, 1)  
 opens[0] = prices[0]  
   
 # Generate realistic volume  
 avg\_volume = 100.0  
 volumes = np.maximum(0.1, np.random.lognormal(np.log(avg\_volume), 0.5, limit))  
   
 df = pd.DataFrame({  
 "open": opens,  
 "high": highs,  
 "low": lows,  
 "close": prices,  
 "volume": volumes  
 }, index=index)  
   
 return df  
   
 except Exception as e:  
 log\_exception("Failed to generate synthetic OHLCV", e)  
 return None  
   
 def \_process\_raw\_ohlcv(self, raw\_data: List[List], symbol: str) -> Optional['pd.DataFrame']:  
 """Process raw OHLCV data from exchange"""  
 if not raw\_data or not HAS\_PANDAS:  
 return None  
   
 try:  
 df = pd.DataFrame(  
 raw\_data,  
 columns=["timestamp", "open", "high", "low", "close", "volume"]  
 )  
   
 # Convert timestamp to datetime index  
 df["timestamp"] = pd.to\_datetime(df["timestamp"], unit="ms", utc=True)  
 df = df.set\_index("timestamp")  
   
 # Ensure numeric types  
 for col in ["open", "high", "low", "close", "volume"]:  
 df[col] = pd.to\_numeric(df[col], errors="coerce")  
   
 # Validate data integrity  
 if not self.\_validate\_ohlcv\_data(df):  
 raise DataValidationError(f"Invalid OHLCV data for {symbol}")  
   
 return df  
   
 except Exception as e:  
 log\_exception("Failed to process raw OHLCV data", e, symbol=symbol)  
 return None  
   
 def \_validate\_ohlcv\_data(self, df: 'pd.DataFrame') -> bool:  
 """Validate OHLCV data integrity"""  
 if df is None or df.empty:  
 return False  
   
 required\_columns = {"open", "high", "low", "close", "volume"}  
 if not required\_columns.issubset(set(df.columns)):  
 return False  
   
 # Check for null values  
 if df[list(required\_columns)].isnull().any().any():  
 return False  
   
 # Check for negative values  
 if (df[list(required\_columns)] < 0).any().any():  
 return False  
   
 # Check OHLC relationships  
 invalid\_ohlc = (  
 (df["high"] < df["low"]) |  
 (df["high"] < df["open"]) |  
 (df["high"] < df["close"]) |  
 (df["low"] > df["open"]) |  
 (df["low"] > df["close"])  
 )  
   
 if invalid\_ohlc.any():  
 return False  
   
 return True  
   
 @retry\_async(max\_attempts=2, base\_delay=0.5)  
 async def get\_ticker(self, symbol: str) -> Optional[Dict[str, Any]]:  
 """Get ticker data with caching"""  
 if not self.ticker\_cb.can\_execute():  
 log\_with\_context("warning", "Ticker fetch blocked by circuit breaker")  
 await asyncio.sleep(0.5)  
 return None  
   
 try:  
 async with self.semaphore:  
 if self.paper\_mode or self.\_exchange is None:  
 price = 30000.0 + random.uniform(-1000, 1000)  
 ticker = {  
 "symbol": symbol,  
 "bid": price - 0.5,  
 "ask": price + 0.5,  
 "last": price,  
 "timestamp": time.time() \* 1000  
 }  
 else:  
 ticker = await self.\_exchange.fetch\_ticker(symbol)  
   
 self.ticker\_cb.record\_success()  
 return ticker  
   
 except Exception as e:  
 self.ticker\_cb.record\_failure()  
 log\_exception("Ticker fetch failed", e, symbol=symbol)  
 return None  
   
 async def close(self) -> None:  
 """Close exchange connection"""  
 if self.\_exchange is not None:  
 try:  
 await self.\_exchange.close()  
 except Exception as e:  
 log\_exception("Error closing exchange connection", e)  
 finally:  
 self.\_exchange = None  
   
 def get\_performance\_stats(self) -> Dict[str, Any]:  
 """Get performance statistics"""  
 if not self.\_request\_times:  
 return {"avg\_request\_time": 0.0, "total\_requests": 0}  
   
 return {  
 "avg\_request\_time": sum(self.\_request\_times) / len(self.\_request\_times),  
 "total\_requests": len(self.\_request\_times),  
 "fetch\_circuit\_breaker": self.fetch\_cb.state,  
 "ticker\_circuit\_breaker": self.ticker\_cb.state  
 }  
  
# =============== Enhanced Technical Indicators ===============  
class TechnicalIndicators:  
 """Optimized technical indicators with proper error handling"""  
   
 @staticmethod  
 def ema(series: 'pd.Series', period: int) -> 'pd.Series':  
 """Exponential Moving Average"""  
 if not HAS\_PANDAS or series is None or series.empty:  
 return pd.Series(dtype=float) if HAS\_PANDAS else None  
   
 try:  
 return series.ewm(span=period, adjust=False).mean()  
 except Exception as e:  
 log\_exception("EMA calculation failed", e, period=period)  
 return pd.Series(index=series.index, dtype=float)  
   
 @staticmethod  
 def rsi(series: 'pd.Series', period: int = 14) -> 'pd.Series':  
 """Relative Strength Index with improved calculation"""  
 if not HAS\_PANDAS or series is None or series.empty or len(series) < period:  
 return pd.Series(dtype=float) if HAS\_PANDAS else None  
   
 try:  
 delta = series.diff()  
 gain = (delta.where(delta > 0, 0)).rolling(window=period, min\_periods=period).mean()  
 loss = (-delta.where(delta < 0, 0)).rolling(window=period, min\_periods=period).mean()  
   
 rs = gain / (loss + 1e-14) # Avoid division by zero  
 rsi = 100 - (100 / (1 + rs))  
   
 return rsi  
   
 except Exception as e:  
 log\_exception("RSI calculation failed", e, period=period)  
 return pd.Series(index=series.index, dtype=float)  
   
 @staticmethod  
 def atr(df: 'pd.DataFrame', period: int = 14) -> 'pd.Series':  
 """Average True Range"""  
 if not HAS\_PANDAS or df is None or df.empty:  
 return pd.Series(dtype=float) if HAS\_PANDAS else None  
   
 try:  
 high\_low = df["high"] - df["low"]  
 high\_close = (df["high"] - df["close"].shift(1)).abs()  
 low\_close = (df["low"] - df["close"].shift(1)).abs()  
   
 true\_range = pd.concat([high\_low, high\_close, low\_close], axis=1).max(axis=1)  
 atr = true\_range.rolling(window=period, min\_periods=period).mean()  
   
 return atr  
   
 except Exception as e:  
 log\_exception("ATR calculation failed", e, period=period)  
 return pd.Series(index=df.index, dtype=float)  
   
 @staticmethod  
 def bollinger\_bands(  
 series: 'pd.Series',   
 period: int = 20,   
 std\_dev: float = 2.0  
 ) -> Tuple['pd.Series', 'pd.Series', 'pd.Series', 'pd.Series']:  
 """Bollinger Bands with squeeze indicator"""  
 if not HAS\_PANDAS or series is None or series.empty:  
 empty\_series = pd.Series(dtype=float) if HAS\_PANDAS else None  
 return empty\_series, empty\_series, empty\_series, empty\_series  
   
 try:  
 rolling\_mean = series.rolling(window=period, min\_periods=period).mean()  
 rolling\_std = series.rolling(window=period, min\_periods=period).std()  
   
 upper\_band = rolling\_mean + (rolling\_std \* std\_dev)  
 lower\_band = rolling\_mean - (rolling\_std \* std\_dev)  
   
 # Squeeze indicator (normalized band width)  
 squeeze = (upper\_band - lower\_band) / (rolling\_mean + 1e-14)  
   
 return rolling\_mean, upper\_band, lower\_band, squeeze  
   
 except Exception as e:  
 log\_exception("Bollinger Bands calculation failed", e, period=period)  
 empty\_series = pd.Series(index=series.index, dtype=float)  
 return empty\_series, empty\_series, empty\_series, empty\_series  
  
# =============== Enhanced Signal Generator ===============  
class SignalGenerator:  
 """Enhanced signal generator with model integration"""  
   
 def \_\_init\_\_(self, config: ConfigDict):  
 self.config = config  
 self.indicators = TechnicalIndicators()  
 self.models: Dict[str, ModelDict] = {}  
 self.\_feature\_cache: Dict[str, Tuple[np.ndarray, float]] = {}  
 self.\_cache\_max\_age = 300 # 5 minutes  
   
 def add\_model(self, name: str, model: Any, model\_type: str) -> None:  
 """Add a prediction model"""  
 try:  
 self.models[name] = {  
 "model": model,  
 "type": model\_type,  
 "warmup\_remaining": 10,  
 "predictions": deque(maxlen=100),  
 "accuracy": 0.0,  
 "last\_used": time.time()  
 }  
 log\_with\_context("info", f"Added model: {name}", model\_type=model\_type)  
 except Exception as e:  
 log\_exception("Failed to add model", e, name=name, model\_type=model\_type)  
   
 def \_extract\_features(self, df: 'pd.DataFrame') -> Optional[np.ndarray]:  
 """Extract features for ML models with caching"""  
 if not HAS\_PANDAS or not HAS\_NUMPY or df is None or df.empty:  
 return None  
   
 # Create cache key  
 cache\_key = f"{id(df)}\_{len(df)}"  
 current\_time = time.time()  
   
 # Check cache  
 if cache\_key in self.\_feature\_cache:  
 features, cache\_time = self.\_feature\_cache[cache\_key]  
 if current\_time - cache\_time < self.\_cache\_max\_age:  
 return features  
   
 try:  
 # Extract price-based features  
 closes = df["close"].astype(float)  
   
 if len(closes) < 60:  
 # Pad if needed  
 closes = pd.Series(  
 np.pad(closes.values, (60 - len(closes), 0), mode="edge"),  
 index=range(60)  
 )  
   
 # Normalize returns  
 returns = closes.pct\_change().fillna(0.0)  
 features = returns.tail(60).values.reshape(1, -1).astype(np.float32)  
   
 # Cache features  
 self.\_feature\_cache[cache\_key] = (features, current\_time)  
   
 # Cleanup old cache entries  
 if len(self.\_feature\_cache) > 100:  
 oldest\_key = min(  
 self.\_feature\_cache.keys(),  
 key=lambda k: self.\_feature\_cache[k][1]  
 )  
 del self.\_feature\_cache[oldest\_key]  
   
 return features  
   
 except Exception as e:  
 log\_exception("Feature extraction failed", e)  
 return None  
   
 def \_predict\_with\_model(  
 self,   
 model\_info: ModelDict,   
 features: np.ndarray  
 ) -> Tuple[Optional[str], float]:  
 """Make prediction with a single model"""  
 if not HAS\_NUMPY:  
 return None, 0.0  
   
 try:  
 model = model\_info["model"]  
 model\_type = model\_info["type"]  
   
 if model\_type == "sklearn" and hasattr(model, "predict\_proba"):  
 proba = model.predict\_proba(features)[0]  
 if len(proba) >= 3: # [hold, buy, sell]  
 idx = np.argmax(proba)  
 confidence = float(proba[idx])  
 action = ["hold", "buy", "sell"][idx]  
 elif len(proba) == 2: # [bearish, bullish]  
 confidence = float(proba[1])  
 action = "buy" if confidence > 0.6 else "sell" if confidence < 0.4 else "hold"  
 else:  
 return None, 0.0  
   
 return action, confidence  
   
 elif hasattr(model, "predict"):  
 prediction = model.predict(features)  
 pred\_value = prediction[0] if isinstance(prediction, (list, np.ndarray)) else prediction  
   
 if isinstance(pred\_value, str):  
 return pred\_value.lower(), 0.6  
 elif isinstance(pred\_value, (int, float)):  
 if pred\_value > 0.1:  
 return "buy", min(float(pred\_value), 1.0)  
 elif pred\_value < -0.1:  
 return "sell", min(abs(float(pred\_value)), 1.0)  
 else:  
 return "hold", 0.5  
   
 return None, 0.0  
   
 except Exception as e:  
 log\_exception("Model prediction failed", e, model\_type=model\_info.get("type"))  
 return None, 0.0  
   
 def \_calculate\_ta\_signals(self, symbol: str, df: 'pd.DataFrame') -> Dict[str, Any]:  
 """Calculate technical analysis signals"""  
 signals\_config = self.config.get("signals", {})  
   
 signal\_result = {  
 "symbol": symbol,  
 "action": "hold",  
 "confidence": 0.0,  
 "score": 0.0,  
 "indicators": {}  
 }  
   
 try:  
 if df is None or df.empty or len(df) < 20:  
 return signal\_result  
   
 close = df["close"]  
   
 # Calculate indicators  
 rsi = self.indicators.rsi(close, signals\_config.get("rsi\_period", 14))  
 bb\_ma, bb\_upper, bb\_lower, bb\_squeeze = self.indicators.bollinger\_bands(  
 close,   
 signals\_config.get("bb\_period", 20),  
 signals\_config.get("bb\_mult", 2.0)  
 )  
 atr = self.indicators.atr(df, 14)  
   
 if rsi is None or bb\_ma is None:  
 return signal\_result  
   
 # Get latest values  
 current\_rsi = rsi.iloc[-1] if not rsi.empty else 50.0  
 current\_price = close.iloc[-1]  
 current\_bb\_upper = bb\_upper.iloc[-1] if not bb\_upper.empty else current\_price  
 current\_bb\_lower = bb\_lower.iloc[-1] if not bb\_lower.empty else current\_price  
 current\_squeeze = bb\_squeeze.iloc[-1] if not bb\_squeeze.empty else 0.1  
   
 # Store indicator values  
 signal\_result["indicators"] = {  
 "rsi": float(current\_rsi),  
 "bb\_position": float((current\_price - current\_bb\_lower) / (current\_bb\_upper - current\_bb\_lower + 1e-14)),  
 "squeeze": float(current\_squeeze),  
 "atr\_ratio": float(atr.iloc[-1] / current\_price) if not atr.empty else 0.0  
 }  
   
 # Generate signals  
 rsi\_buy\_threshold = signals\_config.get("rsi\_buy", 30)  
 rsi\_sell\_threshold = signals\_config.get("rsi\_sell", 70)  
   
 buy\_conditions = [  
 current\_rsi <= rsi\_buy\_threshold,  
 current\_price <= current\_bb\_lower,  
 current\_squeeze < 0.02 # Low volatility breakout potential  
 ]  
   
 sell\_conditions = [  
 current\_rsi >= rsi\_sell\_threshold,  
 current\_price >= current\_bb\_upper  
 ]  
   
 buy\_score = sum(buy\_conditions) / len(buy\_conditions)  
 sell\_score = sum(sell\_conditions) / len(sell\_conditions)  
   
 if buy\_score > 0.6 and sell\_score < 0.3:  
 signal\_result.update({  
 "action": "buy",  
 "confidence": buy\_score,  
 "score": buy\_score  
 })  
 elif sell\_score > 0.6 and buy\_score < 0.3:  
 signal\_result.update({  
 "action": "sell",   
 "confidence": sell\_score,  
 "score": sell\_score  
 })  
 else:  
 signal\_result.update({  
 "action": "hold",  
 "confidence": 0.5,  
 "score": 0.5  
 })  
   
 except Exception as e:  
 log\_exception("TA signal calculation failed", e, symbol=symbol)  
   
 return signal\_result  
   
 def generate\_signal(self, symbol: str, df: 'pd.DataFrame') -> Optional[Dict[str, Any]]:  
 """Generate trading signal with model ensemble"""  
 if not HAS\_PANDAS or df is None or df.empty:  
 return None  
   
 try:  
 # Start with TA signals  
 ta\_signal = self.\_calculate\_ta\_signals(symbol, df)  
   
 weights = self.config.get("weights", {})  
 ta\_weight = weights.get("ta", 0.6)  
 ml\_weight = weights.get("ml", 0.3)  
   
 # Initialize vote aggregation  
 action\_votes = {ta\_signal["action"]: ta\_signal["score"] \* ta\_weight}  
   
 # Add ML model predictions  
 features = self.\_extract\_features(df)  
 if features is not None and self.models:  
 ml\_predictions = []  
   
 for name, model\_info in self.models.items():  
 if model\_info["warmup\_remaining"] > 0:  
 model\_info["warmup\_remaining"] -= 1  
 continue  
   
 action, confidence = self.\_predict\_with\_model(model\_info, features)  
 if action:  
 ml\_predictions.append((action, confidence))  
 model\_info["predictions"].append({  
 "action": action,  
 "confidence": confidence,  
 "timestamp": time.time()  
 })  
 model\_info["last\_used"] = time.time()  
   
 # Ensemble ML predictions  
 if ml\_predictions:  
 for action, confidence in ml\_predictions:  
 weight = confidence \* ml\_weight / len(ml\_predictions)  
 action\_votes[action] = action\_votes.get(action, 0.0) + weight  
   
 # Determine final action  
 if action\_votes:  
 best\_action = max(action\_votes.items(), key=lambda x: x[1])  
 ta\_signal["action"] = best\_action[0]  
 ta\_signal["score"] = best\_action[1]  
 ta\_signal["confidence"] = min(best\_action[1], 1.0)  
   
 return ta\_signal  
   
 except Exception as e:  
 log\_exception("Signal generation failed", e, symbol=symbol)  
 return None  
  
# =============== Enhanced Risk Management ===============  
class RiskManager:  
 """Enhanced risk management with dynamic position sizing"""  
   
 def \_\_init\_\_(self, config: ConfigDict, db\_manager: DatabaseManager):  
 self.config = config  
 self.db\_manager = db\_manager  
   
 # Portfolio state  
 self.equity = 100000.0 # Starting equity for paper trading  
 self.daily\_pnl = 0.0  
 self.drawdown\_peak = self.equity  
 self.\_equity\_lock = threading.RLock()  
   
 # Risk metrics tracking  
 self.returns\_history: deque = deque(maxlen=self.config.get("risk", {}).get("var\_window", 200))  
 self.var\_cache: Optional[float] = None  
 self.var\_last\_calculated = 0.0  
   
 # Position tracking  
 self.position\_tracker: Dict[str, Dict[str, float]] = {}  
   
 def update\_equity(self, new\_equity: float) -> None:  
 """Thread-safe equity update"""  
 with self.\_equity\_lock:  
 old\_equity = self.equity  
 self.equity = float(new\_equity)  
 self.drawdown\_peak = max(self.drawdown\_peak, self.equity)  
   
 # Calculate daily return  
 if old\_equity > 0:  
 daily\_return = (self.equity - old\_equity) / old\_equity  
 self.returns\_history.append(daily\_return)  
   
 # Record metric  
 self.db\_manager.insert\_record("metrics", {  
 "ts": time.time(),  
 "name": "equity",  
 "value": self.equity,  
 "metadata": json.dumps({"daily\_pnl": self.daily\_pnl})  
 })  
   
 def calculate\_position\_size(  
 self,   
 symbol: str,   
 df: 'pd.DataFrame',   
 price: float,  
 signal\_strength: float = 1.0  
 ) -> float:  
 """Calculate position size using ATR and risk-adjusted sizing"""  
 try:  
 risk\_config = self.config.get("risk", {})  
 positioning\_config = self.config.get("positioning", {})  
   
 # Base risk per trade  
 risk\_per\_trade = self.equity \* risk\_config.get("stop\_loss\_pct", 0.02)  
   
 # ATR-based position sizing  
 atr\_period = positioning\_config.get("atr\_period", 14)  
 atr\_multiplier = positioning\_config.get("atr\_risk\_mult", 1.0)  
   
 if HAS\_PANDAS and df is not None and len(df) >= atr\_period:  
 atr\_series = TechnicalIndicators.atr(df, atr\_period)  
 if not atr\_series.empty:  
 current\_atr = atr\_series.iloc[-1]  
 atr\_stop\_distance = current\_atr \* atr\_multiplier  
 else:  
 atr\_stop\_distance = price \* 0.02 # Fallback to 2%  
 else:  
 atr\_stop\_distance = price \* 0.02  
   
 # Calculate base position size  
 if atr\_stop\_distance > 0:  
 base\_size = risk\_per\_trade / atr\_stop\_distance  
 else:  
 base\_size = 0.0  
   
 # Adjust for signal strength  
 adjusted\_size = base\_size \* clamp(signal\_strength, 0.1, 2.0)  
   
 # Apply position limits  
 max\_position\_value = self.equity \* risk\_config.get("max\_single\_pos\_pct", 0.15)  
 max\_size = max\_position\_value / price if price > 0 else 0.0  
   
 final\_size = min(adjusted\_size, max\_size)  
   
 log\_with\_context(  
 "debug",  
 f"Position size calculated: {final\_size:.6f}",  
 symbol=symbol,  
 price=price,  
 atr\_stop\_distance=atr\_stop\_distance,  
 signal\_strength=signal\_strength,  
 risk\_per\_trade=risk\_per\_trade  
 )  
   
 return max(0.0, final\_size)  
   
 except Exception as e:  
 log\_exception("Position size calculation failed", e, symbol=symbol)  
 return 0.0  
   
 def check\_risk\_limits(  
 self,   
 symbol: str,   
 side: str,   
 price: float,   
 quantity: float,  
 open\_positions: PositionDict  
 ) -> Tuple[bool, str]:  
 """Comprehensive risk limit checking"""  
 try:  
 risk\_config = self.config.get("risk", {})  
   
 # Check maximum drawdown  
 current\_drawdown = 1.0 - (self.equity / max(self.drawdown\_peak, 1e-12))  
 max\_drawdown = risk\_config.get("max\_drawdown\_pct", 0.50)  
   
 if current\_drawdown >= max\_drawdown:  
 return False, f"Maximum drawdown exceeded: {current\_drawdown:.2%} >= {max\_drawdown:.2%}"  
   
 # Check daily loss limit  
 daily\_loss\_limit = risk\_config.get("daily\_loss\_limit\_pct", 0.10)  
 if self.daily\_pnl <= -abs(self.equity \* daily\_loss\_limit):  
 return False, f"Daily loss limit exceeded: {self.daily\_pnl:.2f}"  
   
 # Check portfolio exposure for new positions  
 if side.lower() == "buy":  
 position\_value = abs(quantity) \* price  
   
 # Calculate current exposure  
 total\_exposure = sum(  
 abs(pos.get("qty", 0.0)) \* pos.get("avg\_price", 0.0)  
 for pos in open\_positions.values()  
 )  
   
 max\_portfolio\_exposure = risk\_config.get("max\_portfolio\_exposure\_pct", 0.80) \* self.equity  
   
 if total\_exposure + position\_value > max\_portfolio\_exposure:  
 return False, f"Portfolio exposure limit exceeded"  
   
 # Check single position limit  
 max\_single\_position = risk\_config.get("max\_single\_pos\_pct", 0.15) \* self.equity  
 current\_position\_value = open\_positions.get(symbol, {}).get("qty", 0.0) \* price  
   
 if current\_position\_value + position\_value > max\_single\_position:  
 return False, f"Single position limit exceeded for {symbol}"  
   
 # Calculate and check VaR if available  
 var\_limit = self.\_calculate\_var()  
 if var\_limit and hasattr(self, 'portfolio\_var'):  
 if self.portfolio\_var > var\_limit:  
 return False, f"VaR limit exceeded: {self.portfolio\_var:.4f} > {var\_limit:.4f}"  
   
 return True, "Risk checks passed"  
   
 except Exception as e:  
 log\_exception("Risk limit check failed", e, symbol=symbol, side=side)  
 return False, "Risk check error"  
   
 def \_calculate\_var(self, confidence\_level: float = 0.99) -> Optional[float]:  
 """Calculate Value at Risk"""  
 if len(self.returns\_history) < 30:  
 return None  
   
 # Use cached VaR if recent  
 current\_time = time.time()  
 if (self.var\_cache is not None and   
 current\_time - self.var\_last\_calculated < 300): # 5 minutes cache  
 return self.var\_cache  
   
 try:  
 if not HAS\_NUMPY:  
 return None  
   
 returns\_array = np.array(list(self.returns\_history))  
 var\_percentile = (1 - confidence\_level) \* 100  
 var\_value = np.percentile(returns\_array, var\_percentile)  
   
 # Cache the result  
 self.var\_cache = abs(var\_value)  
 self.var\_last\_calculated = current\_time  
   
 # Record VaR metric  
 self.db\_manager.insert\_record("metrics", {  
 "ts": current\_time,  
 "name": "var",  
 "value": self.var\_cache,  
 "metadata": json.dumps({  
 "confidence\_level": confidence\_level,  
 "sample\_size": len(self.returns\_history)  
 })  
 })  
   
 return self.var\_cache  
   
 except Exception as e:  
 log\_exception("VaR calculation failed", e)  
 return None  
   
 def update\_daily\_pnl(self, pnl\_change: float) -> None:  
 """Update daily P&L tracking"""  
 with self.\_equity\_lock:  
 self.daily\_pnl += pnl\_change  
   
 def reset\_daily\_tracking(self) -> None:  
 """Reset daily tracking (call at start of new trading day)"""  
 with self.\_equity\_lock:  
 self.daily\_pnl = 0.0  
  
# =============== Enhanced Order Management ===============  
@dataclass  
class Order:  
 """Enhanced order representation"""  
 id: str  
 timestamp: float  
 symbol: str  
 side: str  
 order\_type: str  
 quantity: float  
 price: float  
 status: str  
 parent\_id: Optional[str] = None  
 expires\_at: Optional[float] = None  
 filled\_quantity: float = 0.0  
 average\_fill\_price: float = 0.0  
 correlation\_id: str = ""  
  
@dataclass   
class Position:  
 """Enhanced position representation"""  
 symbol: str  
 quantity: float  
 average\_price: float  
 unrealized\_pnl: float = 0.0  
 realized\_pnl: float = 0.0  
 last\_updated: float = 0.0  
  
class OrderManager:  
 """Enhanced order management with better tracking"""  
   
 def \_\_init\_\_(self, db\_manager: DatabaseManager, risk\_manager: RiskManager):  
 self.db\_manager = db\_manager  
 self.risk\_manager = risk\_manager  
   
 # Thread-safe data structures  
 self.positions: Dict[str, Position] = {}  
 self.active\_orders: Dict[str, Order] = {}  
 self.\_positions\_lock = threading.RLock()  
 self.\_orders\_lock = threading.RLock()  
   
 # Performance tracking  
 self.execution\_times: deque = deque(maxlen=100)  
   
 def submit\_market\_order(  
 self,   
 symbol: str,   
 side: str,   
 quantity: float,   
 current\_price: float,  
 strategy: str = "core"  
 ) -> Dict[str, Any]:  
 """Submit market order with comprehensive validation"""  
 start\_time = time.time()  
 correlation\_id = getattr(threading.current\_thread(), 'correlation\_id', str(uuid.uuid4()))  
   
 try:  
 # Pre-trade validation  
 with self.\_positions\_lock:  
 positions\_copy = {k: asdict(v) for k, v in self.positions.items()}  
   
 can\_trade, risk\_reason = self.risk\_manager.check\_risk\_limits(  
 symbol, side, current\_price, quantity, positions\_copy  
 )  
   
 if not can\_trade:  
 log\_with\_context(  
 "warning",   
 "Order rejected by risk management",  
 symbol=symbol,   
 side=side,   
 reason=risk\_reason  
 )  
 return {  
 "status": "rejected",  
 "reason": risk\_reason,  
 "order\_id": None  
 }  
   
 # Create order  
 order = Order(  
 id=str(uuid.uuid4()),  
 timestamp=time.time(),  
 symbol=symbol,  
 side=side.lower(),  
 order\_type="market",  
 quantity=abs(quantity),  
 price=current\_price,  
 status="filled", # Immediate fill for paper trading  
 correlation\_id=correlation\_id  
 )  
   
 # Execute order  
 execution\_result = self.\_execute\_order(order, current\_price)  
   
 # Record execution time  
 execution\_time = time.time() - start\_time  
 self.execution\_times.append(execution\_time)  
   
 log\_with\_context(  
 "info",  
 f"Order executed: {side.upper()} {quantity:.6f} {symbol} @ {current\_price:.2f}",  
 order\_id=order.id,  
 execution\_time=f"{execution\_time:.3f}s",  
 strategy=strategy  
 )  
   
 return {  
 "status": "filled",  
 "order\_id": order.id,  
 "executed\_price": current\_price,  
 "executed\_quantity": quantity,  
 "execution\_time": execution\_time  
 }  
   
 except Exception as e:  
 log\_exception("Order submission failed", e, symbol=symbol, side=side)  
 return {  
 "status": "error",   
 "reason": str(e),  
 "order\_id": None  
 }  
   
 def \_execute\_order(self, order: Order, execution\_price: float) -> bool:  
 """Execute order and update positions"""  
 try:  
 with self.\_positions\_lock:  
 # Get or create position  
 if order.symbol not in self.positions:  
 self.positions[order.symbol] = Position(  
 symbol=order.symbol,  
 quantity=0.0,  
 average\_price=0.0,  
 last\_updated=time.time()  
 )  
   
 position = self.positions[order.symbol]  
 old\_quantity = position.quantity  
 old\_avg\_price = position.average\_price  
   
 if order.side == "buy":  
 # Calculate new average price for buys  
 total\_cost = (old\_quantity \* old\_avg\_price) + (order.quantity \* execution\_price)  
 new\_quantity = old\_quantity + order.quantity  
   
 if new\_quantity > 0:  
 position.average\_price = total\_cost / new\_quantity  
 position.quantity = new\_quantity  
   
 elif order.side == "sell":  
 # Calculate realized P&L for sells  
 if old\_quantity > 0:  
 sell\_quantity = min(order.quantity, old\_quantity)  
 realized\_pnl = (execution\_price - old\_avg\_price) \* sell\_quantity  
 position.realized\_pnl += realized\_pnl  
 position.quantity = max(0.0, old\_quantity - order.quantity)  
   
 # Update risk manager with P&L  
 self.risk\_manager.update\_daily\_pnl(realized\_pnl)  
   
 # If position goes negative (short), calculate new average  
 if position.quantity < 0:  
 position.average\_price = execution\_price  
   
 position.last\_updated = time.time()  
   
 # Record trade in database  
 trade\_record = {  
 "id": order.id,  
 "ts": order.timestamp,  
 "symbol": order.symbol,  
 "side": order.side,  
 "qty": order.quantity,  
 "price": execution\_price,  
 "fee": 0.0, # Paper trading  
 "pnl": position.realized\_pnl if order.side == "sell" else 0.0,  
 "strategy": "core",  
 "notes": f"Market order execution",  
 "correlation\_id": order.correlation\_id  
 }  
   
 self.db\_manager.insert\_record("trades", trade\_record)  
   
 # Update position in database  
 position\_record = {  
 "symbol": position.symbol,  
 "qty": position.quantity,  
 "avg\_price": position.average\_price,  
 "sector": "",  
 "region": ""  
 }  
   
 self.db\_manager.insert\_record("positions", position\_record)  
   
 return True  
   
 except Exception as e:  
 log\_exception("Order execution failed", e, order\_id=order.id)  
 return False  
   
 def get\_position(self, symbol: str) -> Optional[Position]:  
 """Get current position for symbol"""  
 with self.\_positions\_lock:  
 return self.positions.get(symbol)  
   
 def get\_all\_positions(self) -> Dict[str, Position]:  
 """Get all current positions"""  
 with self.\_positions\_lock:  
 return self.positions.copy()  
   
 def update\_unrealized\_pnl(self, symbol: str, current\_price: float) -> None:  
 """Update unrealized P&L for position"""  
 with self.\_positions\_lock:  
 if symbol in self.positions:  
 position = self.positions[symbol]  
 if position.quantity != 0:  
 position.unrealized\_pnl = (current\_price - position.average\_price) \* position.quantity  
   
 def get\_portfolio\_value(self, current\_prices: PriceDict) -> float:  
 """Calculate total portfolio value"""  
 with self.\_positions\_lock:  
 total\_value = 0.0  
   
 for symbol, position in self.positions.items():  
 if position.quantity != 0 and symbol in current\_prices:  
 position\_value = position.quantity \* current\_prices[symbol]  
 total\_value += position\_value  
   
 return total\_value  
   
 def get\_performance\_stats(self) -> Dict[str, Any]:  
 """Get order management performance statistics"""  
 if not self.execution\_times:  
 return {"avg\_execution\_time": 0.0, "total\_orders": 0}  
   
 return {  
 "avg\_execution\_time": sum(self.execution\_times) / len(self.execution\_times),  
 "total\_orders": len(self.execution\_times),  
 "active\_positions": len([p for p in self.positions.values() if p.quantity != 0])  
 }  
  
# =============== Enhanced Loki Core ===============  
class LokiCore:  
 """Enhanced main trading system core"""  
   
 def \_\_init\_\_(self, config: ConfigDict):  
 self.config = config  
 self.correlation\_id = str(uuid.uuid4())  
 set\_correlation\_id(self.correlation\_id)  
   
 # Initialize components  
 self.db\_manager = DatabaseManager(DB\_PATH)  
 self.data\_manager = DataManager(  
 max\_symbols=config.get("performance", {}).get("max\_symbols", 100),  
 max\_rows\_per\_symbol=config.get("performance", {}).get("max\_rows\_per\_symbol", 1000)  
 )  
   
 # Create semaphore for concurrency control  
 max\_concurrent = config.get("performance", {}).get("max\_concurrent\_requests", 8)  
 self.semaphore = Semaphore(max\_concurrent)  
   
 # Initialize managers  
 self.exchange = ExchangeManager(config, self.semaphore, self.data\_manager)  
 self.risk\_manager = RiskManager(config, self.db\_manager)  
 self.order\_manager = OrderManager(self.db\_manager, self.risk\_manager)  
 self.signal\_generator = SignalGenerator(config)  
   
 # System state  
 self.running = False  
 self.trading\_task: Optional[asyncio.Task] = None  
 self.\_shutdown\_event = asyncio.Event()  
   
 # Performance monitoring  
 self.start\_time = time.time()  
 self.cycle\_times: deque = deque(maxlen=100)  
 self.error\_count = 0  
 self.max\_consecutive\_errors = 10  
   
 log\_with\_context("info", "Loki Core initialized", correlation\_id=self.correlation\_id)  
   
 async def initialize(self) -> None:  
 """Initialize all system components"""  
 try:  
 # Initialize database  
 self.db\_manager.initialize()  
   
 # Initialize exchange  
 await self.exchange.initialize()  
   
 # Load any saved models  
 await self.\_load\_models()  
   
 log\_with\_context("info", "System initialization completed")  
   
 except Exception as e:  
 log\_exception("System initialization failed", e)  
 raise  
   
 async def \_load\_models(self) -> None:  
 """Load ML models from models directory"""  
 if not MODELS\_DIR.exists():  
 return  
   
 try:  
 model\_files = list(MODELS\_DIR.glob("\*.pkl")) + list(MODELS\_DIR.glob("\*.joblib"))  
   
 for model\_file in model\_files:  
 try:  
 model\_name = model\_file.stem  
 # This is a simplified model loading - in production you'd want more sophisticated model management  
 log\_with\_context("info", f"Found model file: {model\_name}", path=str(model\_file))  
   
 except Exception as e:  
 log\_exception(f"Failed to load model {model\_file}", e)  
   
 except Exception as e:  
 log\_exception("Model loading failed", e)  
   
 async def start\_trading(self) -> bool:  
 """Start the trading system"""  
 if self.running:  
 log\_with\_context("warning", "Trading system already running")  
 return False  
   
 try:  
 await self.initialize()  
   
 self.running = True  
 self.\_shutdown\_event.clear()  
   
 # Start trading loop  
 self.trading\_task = asyncio.create\_task(self.\_trading\_loop())  
   
 log\_with\_context("info", "Trading system started")  
 return True  
   
 except Exception as e:  
 log\_exception("Failed to start trading system", e)  
 self.running = False  
 return False  
   
 async def stop\_trading(self) -> None:  
 """Stop the trading system gracefully"""  
 if not self.running:  
 return  
   
 log\_with\_context("info", "Stopping trading system...")  
   
 self.running = False  
 self.\_shutdown\_event.set()  
   
 # Cancel trading task  
 if self.trading\_task and not self.trading\_task.done():  
 self.trading\_task.cancel()  
 try:  
 await asyncio.wait\_for(self.trading\_task, timeout=10.0)  
 except (asyncio.CancelledError, asyncio.TimeoutError):  
 pass  
   
 # Close resources  
 await self.exchange.close()  
 self.db\_manager.close()  
 self.data\_manager.clear()  
   
 log\_with\_context("info", "Trading system stopped")  
   
 async def \_trading\_loop(self) -> None:  
 """Main trading loop with enhanced error handling"""  
 assets = self.config.get("assets", [])  
 timeframes = self.config.get("timeframes", ["1h"])  
 poll\_interval = self.config.get("poll\_interval", 5.0)  
   
 consecutive\_errors = 0  
   
 while self.running and not self.\_shutdown\_event.is\_set():  
 cycle\_start = time.time()  
   
 try:  
 # Process each asset  
 tasks = []  
 for symbol in assets:  
 task = asyncio.create\_task(  
 self.\_process\_symbol(symbol, timeframes),  
 name=f"process\_{symbol}"  
 )  
 tasks.append(task)  
   
 # Wait for all tasks with timeout  
 try:  
 await asyncio.wait\_for(  
 asyncio.gather(\*tasks, return\_exceptions=True),  
 timeout=poll\_interval \* 2  
 )  
 consecutive\_errors = 0 # Reset error count on success  
   
 except asyncio.TimeoutError:  
 log\_with\_context("warning", "Trading cycle timed out")  
 consecutive\_errors += 1  
   
 # Update cycle time  
 cycle\_time = time.time() - cycle\_start  
 self.cycle\_times.append(cycle\_time)  
   
 # Check for excessive consecutive errors  
 if consecutive\_errors >= self.max\_consecutive\_errors:  
 log\_with\_context("error", "Too many consecutive errors, stopping system")  
 await self.stop\_trading()  
 break  
   
 # Wait for next cycle  
 remaining\_time = max(0, poll\_interval - cycle\_time)  
 if remaining\_time > 0:  
 await asyncio.sleep(remaining\_time)  
   
 # Periodic cleanup  
 if time.time() % 300 < poll\_interval: # Every 5 minutes  
 gc.collect()  
   
 except asyncio.CancelledError:  
 break  
 except Exception as e:  
 consecutive\_errors += 1  
 self.error\_count += 1  
 log\_exception("Trading loop error", e)  
   
 # Exponential backoff on errors  
 error\_delay = min(poll\_interval \* (2 \*\* min(consecutive\_errors, 5)), 60.0)  
 await asyncio.sleep(error\_delay)  
   
 async def \_process\_symbol(self, symbol: str, timeframes: List[str]) -> None:  
 """Process a single trading symbol"""  
 symbol\_correlation\_id = f"{self.correlation\_id}\_{symbol}\_{int(time.time())}"  
 set\_correlation\_id(symbol\_correlation\_id)  
   
 try:  
 # Fetch market data for primary timeframe  
 primary\_timeframe = timeframes[0] if timeframes else "1h"  
   
 df = await self.exchange.fetch\_ohlcv(symbol, primary\_timeframe, limit=256)  
   
 if df is None or df.empty:  
 log\_with\_context("debug", f"No data available for {symbol}")  
 return  
   
 # Generate trading signal  
 signal = self.signal\_generator.generate\_signal(symbol, df)  
   
 if not signal:  
 return  
   
 # Get current price  
 ticker = await self.exchange.get\_ticker(symbol)  
 current\_price = ticker.get("last", df["close"].iloc[-1]) if ticker else df["close"].iloc[-1]  
   
 # Update unrealized P&L  
 self.order\_manager.update\_unrealized\_pnl(symbol, current\_price)  
   
 # Process signal  
 await self.\_execute\_signal(symbol, signal, current\_price, df)  
   
 except Exception as e:  
 log\_exception("Symbol processing failed", e, symbol=symbol)  
   
 async def \_execute\_signal(  
 self,   
 symbol: str,   
 signal: Dict[str, Any],   
 current\_price: float,  
 df: 'pd.DataFrame'  
 ) -> None:  
 """Execute trading signal"""  
 action = signal.get("action", "hold")  
 confidence = signal.get("confidence", 0.0)  
   
 if action == "hold" or confidence < 0.5:  
 return  
   
 try:  
 # Get current position  
 position = self.order\_manager.get\_position(symbol)  
 current\_quantity = position.quantity if position else 0.0  
   
 if action == "buy" and current\_quantity <= 0:  
 # Calculate position size  
 quantity = self.risk\_manager.calculate\_position\_size(  
 symbol, df, current\_price, confidence  
 )  
   
 if quantity > 0:  
 result = self.order\_manager.submit\_market\_order(  
 symbol, "buy", quantity, current\_price  
 )  
   
 if result["status"] == "filled":  
 log\_with\_context(  
 "info",  
 f"BUY order filled",  
 symbol=symbol,  
 quantity=quantity,  
 price=current\_price,  
 confidence=confidence  
 )  
   
 elif action == "sell" and current\_quantity > 0:  
 # Sell current position  
 result = self.order\_manager.submit\_market\_order(  
 symbol, "sell", current\_quantity, current\_price  
 )  
   
 if result["status"] == "filled":  
 log\_with\_context(  
 "info",  
 f"SELL order filled",  
 symbol=symbol,  
 quantity=current\_quantity,  
 price=current\_price,  
 confidence=confidence  
 )  
   
 except Exception as e:  
 log\_exception("Signal execution failed", e, symbol=symbol, action=action)  
   
 def get\_system\_status(self) -> Dict[str, Any]:  
 """Get comprehensive system status"""  
 try:  
 current\_time = time.time()  
 uptime = current\_time - self.start\_time  
   
 # Portfolio summary  
 positions = self.order\_manager.get\_all\_positions()  
 active\_positions = {k: v for k, v in positions.items() if v.quantity != 0}  
   
 # Performance metrics  
 avg\_cycle\_time = sum(self.cycle\_times) / len(self.cycle\_times) if self.cycle\_times else 0.0  
   
 return {  
 "system": {  
 "running": self.running,  
 "uptime\_seconds": uptime,  
 "correlation\_id": self.correlation\_id,  
 "paper\_mode": self.exchange.paper\_mode,  
 "error\_count": self.error\_count  
 },  
 "performance": {  
 "avg\_cycle\_time": avg\_cycle\_time,  
 "total\_cycles": len(self.cycle\_times),  
 "exchange\_stats": self.exchange.get\_performance\_stats(),  
 "order\_stats": self.order\_manager.get\_performance\_stats()  
 },  
 "portfolio": {  
 "equity": self.risk\_manager.equity,  
 "daily\_pnl": self.risk\_manager.daily\_pnl,  
 "active\_positions": len(active\_positions),  
 "positions": {k: asdict(v) for k, v in active\_positions.items()}  
 },  
 "risk": {  
 "current\_drawdown": 1.0 - (self.risk\_manager.equity / max(self.risk\_manager.drawdown\_peak, 1e-12)),  
 "var": self.risk\_manager.\_calculate\_var(),  
 "returns\_samples": len(self.risk\_manager.returns\_history)  
 },  
 "memory": {  
 "data\_manager\_usage": self.data\_manager.get\_memory\_usage(),  
 "total\_datasets": len(self.data\_manager.\_data)  
 }  
 }  
   
 except Exception as e:  
 log\_exception("Failed to get system status", e)  
 return {"error": str(e)}  
  
# =============== Enhanced CLI Interface ===============  
class CLIInterface:  
 """Enhanced command-line interface"""  
   
 def \_\_init\_\_(self, core: LokiCore):  
 self.core = core  
 self.running = True  
   
 def display\_banner(self) -> None:  
 """Display system banner"""  
 banner = """  
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━  
🚀 LOKI TRADING SYSTEM F1 - Enhanced Version 🚀  
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━  
"""  
 print(banner)  
   
 def display\_menu(self) -> None:  
 """Display main menu"""  
 menu = """  
1) 📈 Start Trading System  
2) 🛑 Stop Trading System   
3) 📊 Show System Status  
4) 💼 Show Portfolio  
5) 📈 Show Performance Metrics  
6) 🔧 System Configuration  
7) 🧹 Clear Data Cache  
8) 📋 Export System Report  
9) ❌ Exit  
  
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━  
"""  
 print(menu)  
   
 async def run(self) -> None:  
 """Run the CLI interface"""  
 self.display\_banner()  
   
 while self.running:  
 try:  
 self.display\_menu()  
 choice = input("🔸 Enter your choice: ").strip()  
   
 await self.handle\_choice(choice)  
   
 except KeyboardInterrupt:  
 print("\n👋 Shutdown requested...")  
 await self.handle\_choice("9")  
 break  
 except EOFError:  
 break  
 except Exception as e:  
 log\_exception("CLI error", e)  
 print(f"❌ Error: {e}")  
   
 async def handle\_choice(self, choice: str) -> None:  
 """Handle user menu choice"""  
 try:  
 if choice == "1":  
 await self.\_start\_trading()  
 elif choice == "2":  
 await self.\_stop\_trading()  
 elif choice == "3":  
 await self.\_show\_status()  
 elif choice == "4":  
 await self.\_show\_portfolio()  
 elif choice == "5":  
 await self.\_show\_performance()  
 elif choice == "6":  
 await self.\_show\_configuration()  
 elif choice == "7":  
 await self.\_clear\_cache()  
 elif choice == "8":  
 await self.\_export\_report()  
 elif choice == "9":  
 await self.\_exit()  
 else:  
 print("❓ Invalid choice. Please try again.")  
   
 except Exception as e:  
 log\_exception("Choice handling error", e, choice=choice)  
 print(f"❌ Error handling choice: {e}")  
   
 async def \_start\_trading(self) -> None:  
 """Start trading system"""  
 print("🚀 Starting trading system...")  
 success = await self.core.start\_trading()  
   
 if success:  
 print("✅ Trading system started successfully!")  
 else:  
 print("❌ Failed to start trading system. Check logs for details.")  
   
 async def \_stop\_trading(self) -> None:  
 """Stop trading system"""  
 print("🛑 Stopping trading system...")  
 await self.core.stop\_trading()  
 print("✅ Trading system stopped.")  
   
 async def \_show\_status(self) -> None:  
 """Show system status"""  
 print("\n" + "="\*80)  
 print("📊 SYSTEM STATUS")  
 print("="\*80)  
   
 status = self.core.get\_system\_status()  
   
 # System info  
 system = status.get("system", {})  
 print(f"🟢 Running: {system.get('running', False)}")  
 print(f"📄 Paper Mode: {system.get('paper\_mode', True)}")  
 print(f"⏱️ Uptime: {system.get('uptime\_seconds', 0):.0f} seconds")  
 print(f"❌ Error Count: {system.get('error\_count', 0)}")  
   
 # Performance info  
 performance = status.get("performance", {})  
 print(f"🔄 Avg Cycle Time: {performance.get('avg\_cycle\_time', 0):.3f}s")  
 print(f"📈 Total Cycles: {performance.get('total\_cycles', 0)}")  
   
 # Portfolio info  
 portfolio = status.get("portfolio", {})  
 print(f"💰 Equity: ${portfolio.get('equity', 0):,.2f}")  
 print(f"📊 Daily P&L: ${portfolio.get('daily\_pnl', 0):,.2f}")  
 print(f"🎯 Active Positions: {portfolio.get('active\_positions', 0)}")  
   
 # Risk info  
 risk = status.get("risk", {})  
 print(f"📉 Current Drawdown: {risk.get('current\_drawdown', 0):.2%}")  
 if risk.get('var'):  
 print(f"⚠️ VaR: {risk.get('var', 0):.4f}")  
   
 print("="\*80)  
   
 async def \_show\_portfolio(self) -> None:  
 """Show portfolio details"""  
 print("\n" + "="\*80)  
 print("💼 PORTFOLIO DETAILS")  
 print("="\*80)  
   
 positions = self.core.order\_manager.get\_all\_positions()  
 active\_positions = {k: v for k, v in positions.items() if v.quantity != 0}  
   
 if not active\_positions:  
 print("📭 No active positions")  
 else:  
 print(f"{'Symbol':<12} {'Qty':<12} {'Avg Price':<12} {'Unrealized P&L':<15}")  
 print("-" \* 60)  
   
 total\_unrealized = 0.0  
 for symbol, position in active\_positions.items():  
 print(f"{symbol:<12} {position.quantity:<12.6f} {position.average\_price:<12.2f} ${position.unrealized\_pnl:<15.2f}")  
 total\_unrealized += position.unrealized\_pnl  
   
 print("-" \* 60)  
 print(f"{'Total Unrealized P&L:':<39} ${total\_unrealized:>15.2f}")  
   
 print("="\*80)  
   
 async def \_show\_performance(self) -> None:  
 """Show performance metrics"""  
 print("\n" + "="\*80)  
 print("📈 PERFORMANCE METRICS")  
 print("="\*80)  
   
 status = self.core.get\_system\_status()  
   
 # Exchange performance  
 exchange\_stats = status.get("performance", {}).get("exchange\_stats", {})  
 print("📡 Exchange Performance:")  
 print(f" Average Request Time: {exchange\_stats.get('avg\_request\_time', 0):.3f}s")  
 print(f" Total Requests: {exchange\_stats.get('total\_requests', 0)}")  
 print(f" Fetch Circuit Breaker: {exchange\_stats.get('fetch\_circuit\_breaker', 'UNKNOWN')}")  
   
 # Order performance  
 order\_stats = status.get("performance", {}).get("order\_stats", {})  
 print("\n📋 Order Management:")  
 print(f" Average Execution Time: {order\_stats.get('avg\_execution\_time', 0):.3f}s")  
 print(f" Total Orders: {order\_stats.get('total\_orders', 0)}")  
   
 # Memory usage  
 memory = status.get("memory", {})  
 print("\n💾 Memory Usage:")  
 print(f" Total Datasets: {memory.get('total\_datasets', 0)}")  
   
 usage = memory.get("data\_manager\_usage", {})  
 if usage:  
 total\_memory = sum(usage.values())  
 print(f" Total Memory: {total\_memory:,} bytes")  
 for symbol, mem in usage.items():  
 print(f" {symbol}: {mem:,} bytes")  
   
 print("="\*80)  
   
 async def \_show\_configuration(self) -> None:  
 """Show current configuration"""  
 print("\n" + "="\*80)  
 print("🔧 SYSTEM CONFIGURATION")  
 print("="\*80)  
   
 # Display key configuration sections  
 config\_sections = {  
 "Trading": {  
 "Paper Mode": self.core.config.get("paper", True),  
 "Exchange": self.core.config.get("exchange", "binance"),  
 "Assets": ", ".join(self.core.config.get("assets", [])),  
 "Poll Interval": f"{self.core.config.get('poll\_interval', 5.0)}s"  
 },  
 "Risk Management": self.core.config.get("risk", {}),  
 "Signal Weights": self.core.config.get("weights", {}),  
 "Performance": self.core.config.get("performance", {})  
 }  
   
 for section\_name, section\_data in config\_sections.items():  
 print(f"\n📋 {section\_name}:")  
 if isinstance(section\_data, dict):  
 for key, value in section\_data.items():  
 print(f" {key}: {value}")  
 else:  
 print(f" {section\_data}")  
   
 print("="\*80)  
   
 async def \_clear\_cache(self) -> None:  
 """Clear data cache"""  
 print("🧹 Clearing data cache...")  
   
 # Clear data manager cache  
 self.core.data\_manager.clear()  
   
 # Force garbage collection  
 gc.collect()  
   
 print("✅ Cache cleared successfully!")  
   
 async def \_export\_report(self) -> None:  
 """Export system report"""  
 print("📋 Generating system report...")  
   
 try:  
 # Get comprehensive status  
 status = self.core.get\_system\_status()  
   
 # Add timestamp and additional info  
 report = {  
 "generated\_at": utc\_timestamp(),  
 "generated\_by": "loki\_f1",  
 "system\_status": status,  
 "configuration": self.core.config  
 }  
   
 # Save to file  
 timestamp = int(utc\_timestamp())  
 filename = REPORTS\_DIR / f"loki\_system\_report\_{timestamp}.json"  
   
 with open(filename, 'w', encoding='utf-8') as f:  
 json.dump(report, f, indent=2, default=str)  
   
 print(f"✅ Report exported to: {filename}")  
   
 except Exception as e:  
 log\_exception("Report export failed", e)  
 print(f"❌ Failed to export report: {e}")  
   
 async def \_exit(self) -> None:  
 """Exit the application"""  
 print("👋 Shutting down system...")  
   
 # Stop trading if running  
 if self.core.running:  
 await self.core.stop\_trading()  
   
 self.running = False  
 print("✅ Goodbye!")  
  
# =============== Enhanced Backtesting Engine ===============  
class BacktestEngine:  
 """Memory-efficient backtesting engine"""  
   
 def \_\_init\_\_(self, config: ConfigDict):  
 self.config = config  
 self.results: Dict[str, Any] = {}  
   
 async def run\_backtest(  
 self,   
 symbol: str,   
 timeframe: str = "1h",  
 days: int = 90,  
 initial\_capital: float = 10000.0,  
 fee\_bps: float = 1.0  
 ) -> Dict[str, Any]:  
 """Run comprehensive backtest"""  
 try:  
 log\_with\_context("info", f"Starting backtest for {symbol}", timeframe=timeframe, days=days)  
   
 # Create test exchange manager  
 test\_exchange = ExchangeManager(self.config, Semaphore(1), DataManager())  
 test\_exchange.paper\_mode = True # Force paper mode  
 await test\_exchange.initialize()  
   
 # Fetch historical data  
 limit = max(100, int(days \* (24 if timeframe.endswith("h") else 1)) + 50)  
 df = await test\_exchange.fetch\_ohlcv(symbol, timeframe, limit)  
   
 if df is None or df.empty:  
 return {"success": False, "error": "No data available"}  
   
 # Trim to requested period  
 df = df.tail(days if timeframe == "1d" else days \* 24)  
   
 # Initialize backtest components  
 bt\_risk\_manager = RiskManager(self.config, DatabaseManager(":memory:"))  
 bt\_risk\_manager.equity = initial\_capital  
   
 bt\_signal\_generator = SignalGenerator(self.config)  
   
 # Simulation state  
 cash = initial\_capital  
 position\_qty = 0.0  
 trades = []  
 equity\_curve = []  
 last\_trade\_price = 0.0  
   
 # Run simulation  
 for i, (timestamp, row) in enumerate(df.iterrows()):  
 current\_price = float(row["close"])  
   
 # Record equity  
 portfolio\_value = cash + (position\_qty \* current\_price)  
 equity\_curve.append({  
 "timestamp": timestamp.timestamp() if hasattr(timestamp, 'timestamp') else float(timestamp),  
 "equity": portfolio\_value,  
 "cash": cash,  
 "position\_value": position\_qty \* current\_price  
 })  
   
 # Update equity in risk manager for position sizing  
 bt\_risk\_manager.update\_equity(portfolio\_value)  
   
 # Generate signal using window of data up to current point  
 window\_start = max(0, i - 256)  
 window\_df = df.iloc[window\_start:i+1]  
   
 if len(window\_df) < 20:  
 continue  
   
 signal = bt\_signal\_generator.generate\_signal(symbol, window\_df)  
   
 if not signal:  
 continue  
   
 action = signal.get("action", "hold")  
 confidence = signal.get("confidence", 0.0)  
   
 # Execute trades based on signals  
 if action == "buy" and position\_qty == 0 and confidence > 0.5:  
 # Calculate position size  
 position\_size = bt\_risk\_manager.calculate\_position\_size(  
 symbol, window\_df, current\_price, confidence  
 )  
   
 if position\_size > 0:  
 trade\_value = position\_size \* current\_price  
 fee = trade\_value \* (fee\_bps / 10000)  
   
 if cash >= trade\_value + fee:  
 cash -= (trade\_value + fee)  
 position\_qty = position\_size  
 last\_trade\_price = current\_price  
   
 trades.append({  
 "timestamp": timestamp.timestamp() if hasattr(timestamp, 'timestamp') else float(timestamp),  
 "action": "buy",  
 "price": current\_price,  
 "quantity": position\_size,  
 "fee": fee,  
 "cash\_after": cash  
 })  
   
 elif action == "sell" and position\_qty > 0:  
 # Sell position  
 trade\_value = position\_qty \* current\_price  
 fee = trade\_value \* (fee\_bps / 10000)  
 pnl = (current\_price - last\_trade\_price) \* position\_qty - fee  
   
 cash += (trade\_value - fee)  
   
 trades.append({  
 "timestamp": timestamp.timestamp() if hasattr(timestamp, 'timestamp') else float(timestamp),  
 "action": "sell",  
 "price": current\_price,  
 "quantity": position\_qty,  
 "fee": fee,  
 "pnl": pnl,  
 "cash\_after": cash  
 })  
   
 position\_qty = 0.0  
   
 # Close any remaining position  
 if position\_qty > 0:  
 final\_price = float(df["close"].iloc[-1])  
 trade\_value = position\_qty \* final\_price  
 fee = trade\_value \* (fee\_bps / 10000)  
 pnl = (final\_price - last\_trade\_price) \* position\_qty - fee  
   
 cash += (trade\_value - fee)  
   
 trades.append({  
 "timestamp": df.index[-1].timestamp() if hasattr(df.index[-1], 'timestamp') else float(df.index[-1]),  
 "action": "sell",  
 "price": final\_price,  
 "quantity": position\_qty,  
 "fee": fee,  
 "pnl": pnl,  
 "cash\_after": cash  
 })  
   
 # Calculate metrics  
 final\_equity = cash  
 total\_return = (final\_equity - initial\_capital) / initial\_capital  
   
 # Calculate additional metrics  
 metrics = self.\_calculate\_metrics(equity\_curve, trades)  
   
 results = {  
 "success": True,  
 "symbol": symbol,  
 "timeframe": timeframe,  
 "days": days,  
 "initial\_capital": initial\_capital,  
 "final\_equity": final\_equity,  
 "total\_return": total\_return,  
 "total\_trades": len(trades),  
 "winning\_trades": len([t for t in trades if t.get("pnl", 0) > 0]),  
 "losing\_trades": len([t for t in trades if t.get("pnl", 0) < 0]),  
 "metrics": metrics,  
 "equity\_curve": equity\_curve[-500:], # Limit size  
 "trades": trades[-100:] # Limit size  
 }  
   
 await test\_exchange.close()  
   
 log\_with\_context("info", "Backtest completed",   
 symbol=symbol,   
 total\_return=f"{total\_return:.2%}",  
 total\_trades=len(trades))  
   
 return results  
   
 except Exception as e:  
 log\_exception("Backtest failed", e, symbol=symbol)  
 return {"success": False, "error": str(e)}  
   
 def \_calculate\_metrics(self, equity\_curve: List[Dict], trades: List[Dict]) -> Dict[str, float]:  
 """Calculate performance metrics"""  
 if not equity\_curve or len(equity\_curve) < 2:  
 return {}  
   
 try:  
 if not HAS\_NUMPY:  
 return {"error": "numpy\_required"}  
   
 # Extract equity values  
 equity\_values = np.array([point["equity"] for point in equity\_curve])  
   
 # Calculate returns  
 returns = np.diff(equity\_values) / equity\_values[:-1]  
 returns = returns[~np.isnan(returns)] # Remove NaN values  
   
 if len(returns) == 0:  
 return {}  
   
 # Basic metrics  
 total\_return = (equity\_values[-1] - equity\_values[0]) / equity\_values[0]  
   
 # Volatility (annualized)  
 volatility = np.std(returns) \* np.sqrt(252 \* 24) # Assuming hourly data  
   
 # Sharpe ratio (assuming 0% risk-free rate)  
 sharpe\_ratio = (np.mean(returns) / np.std(returns)) \* np.sqrt(252 \* 24) if np.std(returns) > 0 else 0  
   
 # Maximum drawdown  
 running\_max = np.maximum.accumulate(equity\_values)  
 drawdown = (running\_max - equity\_values) / running\_max  
 max\_drawdown = np.max(drawdown)  
   
 # Sortino ratio  
 downside\_returns = returns[returns < 0]  
 downside\_deviation = np.std(downside\_returns) if len(downside\_returns) > 0 else 0  
 sortino\_ratio = (np.mean(returns) / downside\_deviation) \* np.sqrt(252 \* 24) if downside\_deviation > 0 else 0  
   
 # Win rate  
 profitable\_trades = [t for t in trades if t.get("pnl", 0) > 0]  
 win\_rate = len(profitable\_trades) / len(trades) if trades else 0  
   
 # Average win/loss  
 wins = [t["pnl"] for t in trades if t.get("pnl", 0) > 0]  
 losses = [t["pnl"] for t in trades if t.get("pnl", 0) < 0]  
   
 avg\_win = np.mean(wins) if wins else 0  
 avg\_loss = np.mean(losses) if losses else 0  
 profit\_factor = abs(sum(wins) / sum(losses)) if losses and sum(losses) != 0 else 0  
   
 return {  
 "total\_return": float(total\_return),  
 "volatility": float(volatility),  
 "sharpe\_ratio": float(sharpe\_ratio),  
 "sortino\_ratio": float(sortino\_ratio),  
 "max\_drawdown": float(max\_drawdown),  
 "win\_rate": float(win\_rate),  
 "profit\_factor": float(profit\_factor),  
 "avg\_win": float(avg\_win),  
 "avg\_loss": float(avg\_loss)  
 }  
   
 except Exception as e:  
 log\_exception("Metrics calculation failed", e)  
 return {"error": str(e)}  
  
# =============== Main Application Entry Point ===============  
async def main():  
 """Enhanced main function with proper error handling"""  
 try:  
 # Set main thread correlation ID  
 set\_correlation\_id("MAIN")  
   
 # Load configuration  
 log\_with\_context("info", "Loading configuration...")  
 config = load\_config()  
   
 # Initialize core system  
 log\_with\_context("info", "Initializing Loki Core...")  
 core = LokiCore(config)  
   
 # Initialize CLI  
 cli = CLIInterface(core)  
   
 # Register cleanup handler  
 def cleanup\_handler():  
 """Cleanup on exit"""  
 try:  
 if core.running:  
 # Can't use async in atexit, so just set flag  
 core.running = False  
   
 log\_with\_context("info", "System cleanup completed")  
 except Exception as e:  
 print(f"Cleanup error: {e}")  
   
 atexit.register(cleanup\_handler)  
   
 # Start CLI  
 log\_with\_context("info", "Starting CLI interface...")  
 await cli.run()  
   
 except KeyboardInterrupt:  
 log\_with\_context("info", "Shutdown requested by user")  
 except Exception as e:  
 log\_exception("Fatal error in main", e)  
 sys.exit(1)  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 try:  
 asyncio.run(main())  
 except KeyboardInterrupt:  
 print("\n👋 Goodbye!")  
 except Exception as e:  
 print(f"💥 Fatal error: {e}")  
 sys.exit(1)

🥸🥸🥸🥸🥸🥸🥸🥸🥸🥸🥸🥸

#

# ================= LOKI – APPEND-ONLY PATCH v8.1 ======================

from \_\_future\_\_ import annotations

import threading, queue, gzip, base64

if globals().get("\_LOKI\_V81\_APPLIED", False):

pass

else:

globals()["\_LOKI\_V81\_APPLIED"] = True

V81\_VERSION = "8.1"

# ========================== Config Defaults =========================

# Ensure new config surfaces exist with safe defaults.

try:

L8.config.setdefault("v8", {}).setdefault("integrated", False)

L8.config.setdefault("db", {}).setdefault("table\_prefix", "")

L8.config.setdefault("timeouts", {}).setdefault("pool\_size", 2) # 1..32

L8.config.setdefault("timeouts", {}).setdefault("model\_ms", 600) # already used earlier

L8.config.setdefault("snapshots", {}).setdefault("max\_snapshot\_size\_mb", 10)

L8.config.setdefault("snapshots", {}).setdefault("max\_records", 1000)

L8.config.setdefault("snapshots", {}).setdefault("compress", False)

L8.\_save\_config()

except Exception:

pass

# =================== Config Validator Augmentation ==================

# Merge paper-mode & resource limits into main validator output.

if "\_v8\_config\_validate" in globals():

\_v8\_cfg\_val\_prev = \_v8\_config\_validate # keep reference

def \_v8\_config\_validate():

res = \_v8\_cfg\_val\_prev()

try:

cfg = L8.config or {}

# ---- v8.integrated (feature gate)

integrated = bool(((cfg.get("v8") or {}).get("integrated", False)))

if not isinstance(integrated, bool):

res["warnings"].append("v8.integrated must be boolean")

res["ok"] = False

# ---- timeouts.pool\_size

try:

ps = int((cfg.get("timeouts") or {}).get("pool\_size", 2))

if ps < 1 or ps > 32:

res["warnings"].append("timeouts.pool\_size must be in [1, 32]")

res["ok"] = False

except Exception:

res["warnings"].append("timeouts.pool\_size malformed")

res["ok"] = False

# ---- snapshots caps

snap = cfg.get("snapshots") or {}

try:

sz = float(snap.get("max\_snapshot\_size\_mb", 10))

if sz <= 0 or sz > 1024:

res["warnings"].append("snapshots.max\_snapshot\_size\_mb should be in (0, 1024]")

res["ok"] = False

except Exception:

res["warnings"].append("snapshots.max\_snapshot\_size\_mb malformed")

res["ok"] = False

try:

mr = int(snap.get("max\_records", 1000))

if mr < 10:

res["warnings"].append("snapshots.max\_records too low (<10)")

res["ok"] = False

except Exception:

res["warnings"].append("snapshots.max\_records malformed")

res["ok"] = False

if not isinstance(snap.get("compress", False), bool):

res["warnings"].append("snapshots.compress must be boolean")

res["ok"] = False

# ---- paper-mode defaults already set by v8 reliability layer;

# re-assert presence without failing.

paper = cfg.get("paper") or {}

for k in ("baseline\_price", "slippage\_bps", "latency\_ms"):

if k not in paper:

res["warnings"].append(f"config.paper.{k} missing; default assumed")

res["ok"] = False

except Exception:

pass

# Add our version marker

res.setdefault("versions", {})["v8.1"] = V81\_VERSION

return res

globals()["\_v8\_config\_validate"] = \_v8\_config\_validate

# ============== Feature Gate: Reduce Invasiveness When Set ==========

# If v8.integrated is True, newer wrappers/adapters will no-op.

def \_v81\_feature\_gate\_enabled() -> bool:

try:

return bool((L8.config.get("v8") or {}).get("integrated", False))

except Exception:

return False

# ==================== Bounded Prediction Queue ======================

# Replaces unbounded/executor-style submission with a bounded queue and N workers.

class \_V81PredictQueue:

def \_\_init\_\_(self, max\_workers: int = 2, capacity: int = None):

self.max\_workers = max(1, min(32, int(max\_workers or 2)))

self.capacity = int(capacity or (self.max\_workers \* 4))

self.q: "queue.Queue[tuple]" = queue.Queue(maxsize=self.capacity)

self.\_workers: list[threading.Thread] = []

self.\_stop = threading.Event()

self.\_started = False

def \_worker(self):

while not self.\_stop.is\_set():

try:

fn, args, kwargs, result\_box = self.q.get(timeout=0.1)

except queue.Empty:

continue

try:

if self.\_stop.is\_set():

result\_box["exc"] = RuntimeError("predict queue stopping")

else:

result\_box["ret"] = fn(\*args, \*\*kwargs)

except Exception as e:

result\_box["exc"] = e

finally:

self.q.task\_done()

def start(self):

if self.\_started:

return

self.\_stop.clear()

for \_ in range(self.max\_workers):

t = threading.Thread(target=self.\_worker, name="v81-predict", daemon=True)

t.start()

self.\_workers.append(t)

self.\_started = True

def stop(self, join\_timeout: float = 0.5):

if not self.\_started:

return

self.\_stop.set()

# Drain queue fast

try:

while True:

self.q.get\_nowait()

self.q.task\_done()

except queue.Empty:

pass

for t in self.\_workers:

t.join(timeout=join\_timeout)

self.\_workers.clear()

self.\_started = False

def submit(self, fn, \*args, \*\*kwargs):

if not self.\_started:

self.start()

result\_box = {"ret": None, "exc": None}

try:

self.q.put\_nowait((fn, args, kwargs, result\_box))

return result\_box

except queue.Full:

# Reject overload; caller must handle fallback

raise TimeoutError("prediction queue overloaded")

# Create singleton predict queue sized by config

try:

\_V81\_PREDICT\_QUEUE = \_V81PredictQueue(

max\_workers=int((L8.config.get("timeouts") or {}).get("pool\_size", 2)),

capacity=None

)

except Exception:

\_V81\_PREDICT\_QUEUE = \_V81PredictQueue(2, None)

# Replace timeout call with queue + manual timeout wait

def timeout\_call\_v81(fn, timeout\_ms: int, \*args, \*\*kwargs):

"""

Submit to bounded queue; wait up to timeout\_ms.

On overload or timeout, raise TimeoutError so caller can fallback to TA/ML.

"""

if \_v81\_feature\_gate\_enabled():

# If integrated, do not change behavior; defer to original timeout\_call if present

if "timeout\_call" in globals():

return timeout\_call(fn, timeout\_ms, \*args, \*\*kwargs) # type: ignore

try:

box = \_V81\_PREDICT\_QUEUE.submit(fn, \*args, \*\*kwargs)

except TimeoutError:

try: L8.write\_alert("MODEL\_QUEUE\_OVERLOAD: falling back to TA/ML")

except Exception: pass

raise

# Busy-wait (fine for bounded timeouts) with short sleeps

end = time.time() + (timeout\_ms / 1000.0)

while time.time() < end:

if box["exc"] is not None:

raise box["exc"]

if box["ret"] is not None:

return box["ret"]

time.sleep(0.002)

# Timed out

try: L8.write\_alert("MODEL\_TIMEOUT: prediction timed out (v8.1)")

except Exception: pass

raise TimeoutError("model prediction timed out (v8.1)")

# Re-wrap model prediction path to use timeout\_call\_v81 (supersedes v8 wrapper)

def \_v81\_wrap\_model\_predict\_timeout():

if \_v81\_feature\_gate\_enabled():

return

for cls\_name in ("Strategy", "Model", "SignalGenerator"):

CLS = globals().get(cls\_name)

if not CLS or getattr(CLS, "\_v81\_pred\_to", False):

continue

pred = getattr(CLS, "\_predict\_with\_model", None)

if not callable(pred):

continue

def \_predict\_with\_model(self, \*a, \*\*kw):

try:

ms = int((L8.config.get("timeouts", {}) or {}).get("model\_ms", 600))

except Exception:

ms = 600

try:

return timeout\_call\_v81(lambda: pred(self, \*a, \*\*kw), ms)

except TimeoutError:

# Fallback: neutral → rely on TA/ML only

try: LLM\_BREAKER.record\_timeout()

except Exception: pass

return None

CLS.\_predict\_with\_model = \_predict\_with\_model

CLS.\_v81\_pred\_to = True

\_v81\_wrap\_model\_predict\_timeout()

# Ensure graceful stop of prediction queue on exit

try:

atexit.register(lambda: \_V81\_PREDICT\_QUEUE.stop())

except Exception:

pass

# ===================== Snapshot Caps & Compression ==================

# Wrap/replace the snapshot writer/reader with capped+compressed versions.

# Resolve table names (allow optional prefix)

def \_v81\_tbl(name: str) -> str:

try:

prefix = (L8.config.get("db") or {}).get("table\_prefix", "")

except Exception:

prefix = ""

return f"{prefix}{name}"

# Create tables if not present (prefixed)

def \_v81\_db\_ensure\_tables():

DBM = globals().get("DatabaseManager") or globals().get("DBManager") or globals().get("DbManager") or globals().get("DB\_Manager")

if not DBM:

return

try:

db = DBM()

with DBTransaction(db) as tx:

tx.exec(f"""CREATE TABLE IF NOT EXISTS {\_v81\_tbl("executions")} (

order\_id TEXT NOT NULL,

fill\_seq INTEGER NOT NULL,

symbol TEXT,

qty REAL,

price REAL,

ts INTEGER,

PRIMARY KEY(order\_id, fill\_seq)

)""")

tx.exec(f"""CREATE TABLE IF NOT EXISTS {\_v81\_tbl("positions\_snapshot")} (

ts INTEGER NOT NULL,

payload TEXT NOT NULL

)""")

except Exception as e:

try: L8.write\_alert(f"DB\_BOOTSTRAP\_V81\_ERR: {e}")

except Exception: pass

\_v81\_db\_ensure\_tables()

def \_v81\_snapshot\_positions\_impl():

DBM = globals().get("DatabaseManager") or globals().get("DBManager") or globals().get("DbManager") or globals().get("DB\_Manager")

if not DBM:

return

state = \_v8\_collect\_positions\_state() if "\_v8\_collect\_positions\_state" in globals() else {}

cfg = L8.config or {}

s\_cfg = cfg.get("snapshots", {})

compress = bool(s\_cfg.get("compress", False))

max\_mb = float(s\_cfg.get("max\_snapshot\_size\_mb", 10))

max\_records = int(s\_cfg.get("max\_records", 1000))

# serialize

raw = json.dumps(state).encode("utf-8")

if compress:

payload\_bytes = gzip.compress(raw)

payload = json.dumps({

"\_compressed": True,

"\_encoding": "gzip+base64",

"data": base64.b64encode(payload\_bytes).decode("ascii")

})

size\_bytes = len(payload\_bytes)

else:

payload = json.dumps(state)

size\_bytes = len(raw)

# size cap

if (size\_bytes / (1024 \* 1024)) > max\_mb:

try: L8.write\_alert(f"SNAPSHOT\_SKIPPED: size {size\_bytes}B exceeds cap {max\_mb}MB")

except Exception: pass

return

# rotate if necessary

db = DBM()

try:

with DBTransaction(db) as tx:

# delete oldest if count >= max\_records

try:

rows = db.query\_records(f"SELECT COUNT(1) as c FROM {\_v81\_tbl('positions\_snapshot')}", ())

cnt = int(list(rows[0].values())[0]) if rows else 0

except Exception:

cnt = 0

if cnt >= max\_records:

tx.exec(f"""

DELETE FROM {\_v81\_tbl('positions\_snapshot')}

WHERE ts IN (

SELECT ts FROM {\_v81\_tbl('positions\_snapshot')} ORDER BY ts ASC LIMIT ?

)

""", (cnt - max\_records + 1,))

tx.exec(f"INSERT INTO {\_v81\_tbl('positions\_snapshot')}(ts, payload) VALUES (?,?)",

(int(time.time()), payload))

except Exception as e:

try: L8.write\_alert(f"SNAPSHOT\_DB\_V81\_ERR: {e}")

except Exception: pass

# Override the previous snapshot writer with capped+compressed version

globals()["\_v8\_snapshot\_positions"] = \_v81\_snapshot\_positions\_impl

def \_v81\_restore\_positions\_state():

DBM = globals().get("DatabaseManager") or globals().get("DBManager") or globals().get("DbManager") or globals().get("DB\_Manager")

if not DBM:

return {"status":"no\_db"}

db = DBM()

try:

rows = db.query\_records(f"SELECT payload FROM {\_v81\_tbl('positions\_snapshot')} ORDER BY ts DESC LIMIT 1", ())

if not rows:

return {"status":"empty"}

payload = list(rows[0].values())[0]

try:

obj = json.loads(payload)

if isinstance(obj, dict) and obj.get("\_compressed") and obj.get("encoding", obj.get("\_encoding")):

# v8.1 compressed format

b64 = obj.get("data", "")

raw = gzip.decompress(base64.b64decode(b64.encode("ascii")))

return {"status":"ok", "payload": json.loads(raw.decode("utf-8")), "compressed": True}

else:

# plain JSON

return {"status":"ok", "payload": obj, "compressed": False}

except Exception:

# payload may be plain json string (older), try direct

return {"status":"ok", "payload": json.loads(payload), "compressed": False}

except Exception as e:

try: L8.write\_alert(f"RESTORE\_V81\_ERR: {e}")

except Exception: pass

return {"status":"error", "error": str(e)}

globals()["\_v8\_restore\_positions\_state"] = \_v81\_restore\_positions\_state

# Healthcheck: add capacity info & version

def \_v81\_health\_extend():

Ext = globals().get("\_CommandRouterExt")

if not Ext or getattr(Ext, "\_v81\_health", False):

return

orig = Ext.handle

async def handle(self, cmd: str, \*args: str) -> str:

c = (cmd or "").strip().lower()

if c == "healthcheck":

base = await orig(self, cmd, \*args)

add = {

"v8.1": V81\_VERSION,

"predict\_queue": {

"workers": \_V81\_PREDICT\_QUEUE.max\_workers,

"capacity": \_V81\_PREDICT\_QUEUE.capacity

},

"snapshots": {

"compress": bool((L8.config.get("snapshots") or {}).get("compress", False)),

"max\_mb": float((L8.config.get("snapshots") or {}).get("max\_snapshot\_size\_mb", 10)),

"max\_records": int((L8.config.get("snapshots") or {}).get("max\_records", 1000))

},

"feature\_gate\_integrated": \_v81\_feature\_gate\_enabled()

}

try:

obj = json.loads(base); obj.update(add)

return json.dumps(obj, indent=2)

except Exception:

return json.dumps(add, indent=2)

return await orig(self, cmd, \*args)

Ext.handle = handle

Ext.\_v81\_health = True

\_v81\_health\_extend()

# Optional: expose simple router commands if you have router fallback

def \_v81\_extend\_router():

Ext = globals().get("\_CommandRouterExt")

if not Ext or getattr(Ext, "\_v81\_router2", False):

return

orig = Ext.handle

async def handle(self, cmd: str, \*args: str) -> str:

c = (cmd or "").strip().lower()

if c == "version":

# Chain versions

return f"Loki v8.1"

if c == "config\_validate":

return json.dumps(\_v8\_config\_validate(), indent=2)

return await orig(self, cmd, \*args)

Ext.handle = handle

Ext.\_v81\_router2 = True

\_v81\_extend\_router()

# Boot note

if \_\_name\_\_ == "\_\_main\_\_" and globals().get("\_LOKI\_PATCH\_MAIN\_OK", False) and not globals().get("\_LOKI\_V81\_BOOTMSG", False):

globals()["\_LOKI\_V81\_BOOTMSG"] = True

print("[v8.1 active] bounded prediction queue, snapshot caps/compression, unified config validation")