api/**init**.py

```python
# api/__init__.py
"""
Enhanced API package for the cryptocurrency matching engine.

This package provides a comprehensive Flask-based REST API with
WebSocket support
for real-time market data streaming. Features include:

- Order submission and cancellation
- Real-time market data feeds
- Connection management and rate limiting
- Comprehensive error handling and logging
- Performance monitoring and statistics

Usage:
    from api.app import create_app, run

    app = create_app()
    run(host="0.0.0.0", port=5000)
"""

from .app import create_app, run, engine, socketio

__version__ = "2.0.0"
__all__ = ["create_app", "run", "engine", "socketio"]
```

api/app.py

```python
# api/app.py
# CRITICAL: eventlet.monkey_patch() must be called FIRST, before any
other imports
import eventlet
eventlet.monkey_patch()

import logging
import time
import json
from flask import Flask, request, jsonify
from flask_socketio import SocketIO, emit, disconnect
from decimal import Decimal, InvalidOperation
from typing import Dict, Any, Optional
import threading
from functools import wraps
from collections import defaultdict, deque
import os
import sys

# Add the parent directory to the Python path to find the engine
```

```
module
sys.path.insert(0,
os.path.dirname(os.path.dirname(os.path.abspath(__file__))))

# Now import the engine modules
from engine.matcher import MatchingEngine, create_crypto_engine,
MatchingEngineError
from engine.book import Trade
from engine.order import Order


# Configure logging
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s %(levelname)s [%(name)s] %(message)s',
    handlers=[
        logging.StreamHandler(),
        logging.FileHandler('matching_engine.log')
    ]
)
logger = logging.getLogger("matching-api")

# Create app & socketio
# Replace this section in your api/app.py (around line 40-50):

# Create app & socketio
app = Flask(__name__)
app.config.update({
    "SECRET_KEY": "your-secret-key-change-in-production",
    "DEBUG": False,
    "TESTING": False
})

# Configure static files IMMEDIATELY after creating the Flask app
static_dir =
os.path.join(os.path.dirname(os.path.dirname(os.path.abspath(__file__)
)), 'frontend')
app.static_folder = static_dir
app.static_url_path = ''

print(f"Static folder configured: {app.static_folder}")  # Debug line
to verify path

socketio = SocketIO(
    app,
    cors_allowed_origins="*",
    async_mode="eventlet",
    logger=True,
    engineio_logger=False
)
```

```
# Create matching engine
engine = create_crypto_engine()

# Connection management
class ConnectionManager:
    """Manages WebSocket connections and subscriptions."""

    def __init__(self):
        self.connections: Dict[str, Dict] = {}  # session_id ->
connection_info
        self.symbol_subscribers: Dict[str, set] = defaultdict(set)  #
symbol -> set of session_ids
        self.trade_subscribers: set = set()  # session_ids subscribed
to trades
        self.lock = threading.RLock()

    def add_connection(self, session_id: str, user_info: Dict = None):
        """Add a new connection."""
        with self.lock:
            self.connections[session_id] = {
                "connected_at": time.time(),
                "user_info": user_info or {},
                "subscribed_symbols": set(),
                "subscribed_to_trades": False,
                "message_count": 0
            }

    def remove_connection(self, session_id: str):
        """Remove connection and clean up subscriptions."""
        with self.lock:
            if session_id in self.connections:
                # Clean up symbol subscriptions
                for symbol in self.connections[session_id]
["subscribed_symbols"]:

self.symbol_subscribers[symbol].discard(session_id)

                # Clean up trade subscription
                self.trade_subscribers.discard(session_id)

                del self.connections[session_id]

    def subscribe_to_symbol(self, session_id: str, symbol: str):
        """Subscribe connection to symbol updates."""
        with self.lock:
            if session_id in self.connections:
                self.symbol_subscribers[symbol].add(session_id)
                self.connections[session_id]
```

```
["subscribed_symbols"].add(symbol)

    def unsubscribe_from_symbol(self, session_id: str, symbol: str):
        """Unsubscribe connection from symbol updates."""
        with self.lock:
            self.symbol_subscribers[symbol].discard(session_id)
            if session_id in self.connections:
                self.connections[session_id]
["subscribed_symbols"].discard(symbol)

    def subscribe_to_trades(self, session_id: str):
        """Subscribe connection to trade feed."""
        with self.lock:
            self.trade_subscribers.add(session_id)
            if session_id in self.connections:
                self.connections[session_id]["subscribed_to_trades"] =
True

    def get_symbol_subscribers(self, symbol: str) -> set:
        """Get all subscribers for a symbol."""
        with self.lock:
            return self.symbol_subscribers[symbol].copy()

    def get_trade_subscribers(self) -> set:
        """Get all trade subscribers."""
        with self.lock:
            return self.trade_subscribers.copy()

    def get_connection_count(self) -> int:
        """Get total connection count."""
        with self.lock:
            return len(self.connections)

# Global connection manager
conn_mgr = ConnectionManager()

# Rate limiting
class RateLimiter:
    """Simple token bucket rate limiter."""

    def __init__(self, max_requests: int = 10000, window_seconds: int
= 60):
        self.max_requests = max_requests
        self.window_seconds = window_seconds
        self.requests: Dict[str, deque] = defaultdict(deque)
        self.lock = threading.RLock()

    def is_allowed(self, identifier: str) -> bool:
        """Check if request is allowed for identifier."""
```

```
        with self.lock:
            now = time.time()
            window_start = now - self.window_seconds

            # Clean old requests
            while self.requests[identifier] and
self.requests[identifier][0] < window_start:
                self.requests[identifier].popleft()

            # Check limit
            if len(self.requests[identifier]) >= self.max_requests:
                return False

            # Add current request
            self.requests[identifier].append(now)
            return True

# Global rate limiter
rate_limiter = RateLimiter(max_requests=10000, window_seconds=60)

def rate_limit_required(f):
    """Rate limiting decorator."""
    @wraps(f)
    def decorated_function(*args, **kwargs):
        # Use IP address as identifier
        identifier = request.environ.get('REMOTE_ADDR', 'unknown')

        if not rate_limiter.is_allowed(identifier):
            logger.warning(f"Rate limit exceeded for {identifier}")
            return jsonify({
                "error": "Rate limit exceeded",
                "error_code": "RATE_LIMIT",
                "retry_after": 60
            }), 429

        return f(*args, **kwargs)
    return decorated_function

def validate_json_request(required_fields: list = None):
    """Decorator to validate JSON requests."""
    def decorator(f):
        @wraps(f)
        def decorated_function(*args, **kwargs):
            if not request.is_json:
                return jsonify({
                    "error": "Content-Type must be application/json",
                    "error_code": "INVALID_CONTENT_TYPE"
                }), 400

            data = request.get_json()
```

```
            if not data:
                return jsonify({
                    "error": "Invalid JSON payload",
                    "error_code": "INVALID_JSON"
                }), 400

            if required_fields:
                missing_fields = [field for field in required_fields
if field not in data]
                if missing_fields:
                    return jsonify({
                        "error": f"Missing required fields:
{missing_fields}",
                        "error_code": "MISSING_FIELDS"
                    }), 400

            return f(*args, **kwargs)
        return decorated_function
    return decorator

# Event handlers for engine
def handle_trade(trade: Trade):
    """Handle trade events from engine."""
    trade_data = trade.to_dict()

    # Emit to trade subscribers
    subscribers = conn_mgr.get_trade_subscribers()
    if subscribers:
        socketio.emit("trade", trade_data, namespace="/market")
        logger.debug(f"Emitted trade to {len(subscribers)}
subscribers")

def handle_order_event(order: Order, event_type: str):
    """Handle order events from engine."""
    order_data = order.to_dict()
    order_data["event_type"] = event_type

    # Emit to symbol subscribers
    subscribers = conn_mgr.get_symbol_subscribers(order.symbol)
    if subscribers:
        socketio.emit("order_event", order_data, namespace="/market")
        logger.debug(f"Emitted order event to {len(subscribers)}
subscribers")

# Register event handlers
engine.add_trade_handler(handle_trade)
engine.add_order_handler(handle_order_event)

# ---- Static File Route for Frontend ----
@app.route("/")
```

```python
def index():
    """Serve the frontend."""
    return app.send_static_file('index.html')

@app.route("/<path:filename>")
def static_files(filename):
    """Serve static files."""
    return app.send_static_file(filename)

# ---- REST API Endpoints ----

@app.route("/health", methods=["GET"])
def health():
    """Health check endpoint."""
    return jsonify({
        "status": "healthy",
        "timestamp": time.time(),
        "uptime": time.time() - engine.metrics["start_time"],
        "version": "2.0.0",
        "connections": conn_mgr.get_connection_count()
    }), 200

@app.route("/symbols", methods=["GET"])
def get_symbols():
    """Get list of supported trading symbols."""
    try:
        symbols = engine.get_symbols()
        symbol_info = {}

        for symbol in symbols:
            bbo = engine.get_bbo(symbol)
            stats = engine.get_statistics(symbol)
            symbol_info[symbol] = {
                "bbo": bbo,
                "active_orders": stats.get("active_orders", 0),
                "total_trades": stats.get("total_trades", 0)
            }

        return jsonify({
            "symbols": symbols,
            "symbol_info": symbol_info,
            "count": len(symbols)
        }), 200

    except Exception as e:
        logger.exception("Error fetching symbols")
        return jsonify({
            "error": "Failed to fetch symbols",
            "detail": str(e)
        }), 500
```

```python
@app.route("/order", methods=["POST"])
@rate_limit_required
@validate_json_request(required_fields=["symbol", "side",
"order_type", "quantity"])
def submit_order():
    """Submit a new order."""
    try:
        data = request.get_json()

        # Extract parameters
        symbol = data["symbol"]
        side = data["side"]
        order_type = data["order_type"]
        quantity = data["quantity"]
        price = data.get("price")
        client_order_id = data.get("client_order_id")

        logger.info(f"Order submission: {symbol} {side} {order_type}
qty={quantity} price={price}")

        # Submit order to engine
        result = engine.submit_order(
            symbol=symbol,
            side=side,
            order_type=order_type,
            quantity=quantity,
            price=price,
            client_order_id=client_order_id
        )

        # Check for errors
        if "error" in result:
            status_code = 400 if result.get("error_code") ==
"VALIDATION_ERROR" else 500
            return jsonify(result), status_code

        # Emit market data updates
        try:
            # Emit L2 update to symbol subscribers
            snapshot = engine.get_book_snapshot(symbol, levels=10)
            subscribers = conn_mgr.get_symbol_subscribers(symbol)
            if subscribers:
                socketio.emit("l2_update", snapshot,
namespace="/market")
        except Exception as e:
            logger.warning(f"Failed to emit market data update: {e}")

        return jsonify(result), 200
```

```
    except MatchingEngineError as e:
        logger.warning(f"Matching engine error: {e}")
        return jsonify({
            "error": str(e),
            "error_code": "ENGINE_ERROR"
        }), 400

    except Exception as e:
        logger.exception("Unexpected error in order submission")
        return jsonify({
            "error": "Internal server error",
            "error_code": "INTERNAL_ERROR"
        }), 500

@app.route("/cancel", methods=["POST"])
@rate_limit_required
@validate_json_request(required_fields=["symbol"])
def cancel_order():
    """Cancel an existing order."""
    try:
        data = request.get_json()

        symbol = data["symbol"]
        order_id = data.get("order_id")
        client_order_id = data.get("client_order_id")

        if not order_id and not client_order_id:
            return jsonify({
                "error": "Must provide order_id or client_order_id",
                "error_code": "MISSING_ORDER_ID"
            }), 400

        logger.info(f"Cancel request: symbol={symbol}
order_id={order_id} client_order_id={client_order_id}")

        result = engine.cancel_order(symbol, order_id,
client_order_id)

        # Emit market data update if successful
        if result.get("success"):
            try:
                snapshot = engine.get_book_snapshot(symbol, levels=10)
                subscribers = conn_mgr.get_symbol_subscribers(symbol)
                if subscribers:
                    socketio.emit("l2_update", snapshot,
namespace="/market")
            except Exception as e:
                logger.warning(f"Failed to emit market data update
```

```
after cancel: {e}")

        status_code = 200 if result.get("success") else 404
        return jsonify(result), status_code

    except Exception as e:
        logger.exception("Error in order cancellation")
        return jsonify({
            "error": "Failed to cancel order",
            "detail": str(e),
            "error_code": "CANCEL_ERROR"
        }), 500

@app.route("/order/<order_id>", methods=["GET"])
def get_order_status(order_id):
    """Get order status by order_id."""
    try:
        symbol = request.args.get("symbol")
        if not symbol:
            return jsonify({
                "error": "Symbol parameter is required",
                "error_code": "MISSING_SYMBOL"
            }), 400

        status = engine.get_order_status(symbol, order_id=order_id)

        if not status:
            return jsonify({
                "error": "Order not found",
                "error_code": "ORDER_NOT_FOUND"
            }), 404

        return jsonify(status), 200

    except Exception as e:
        logger.exception("Error fetching order status")
        return jsonify({
            "error": "Failed to fetch order status",
            "detail": str(e)
        }), 500

@app.route("/book/<symbol>", methods=["GET"])
def get_order_book(symbol):
    """Get order book snapshot."""
    try:
        levels = int(request.args.get("levels", 10))
        levels = min(max(levels, 1), 100)  # Limit between 1-100

        snapshot = engine.get_book_snapshot(symbol, levels)
```

```
        return jsonify(snapshot), 200

    except ValueError:
        return jsonify({
            "error": "Invalid levels parameter",
            "error_code": "INVALID_PARAMETER"
        }), 400

    except Exception as e:
        logger.exception("Error fetching order book")
        return jsonify({
            "error": "Failed to fetch order book",
            "detail": str(e)
        }), 500

@app.route("/bbo/<symbol>", methods=["GET"])
def get_best_bid_offer(symbol):
    """Get best bid/offer for symbol."""
    try:
        bbo = engine.get_bbo(symbol)
        return jsonify(bbo), 200

    except Exception as e:
        logger.exception("Error fetching BBO")
        return jsonify({
            "error": "Failed to fetch BBO",
            "detail": str(e)
        }), 500

@app.route("/statistics", methods=["GET"])
def get_statistics():
    """Get engine statistics."""
    try:
        symbol = request.args.get("symbol")
        stats = engine.get_statistics(symbol)
        return jsonify(stats), 200

    except Exception as e:
        logger.exception("Error fetching statistics")
        return jsonify({
            "error": "Failed to fetch statistics",
            "detail": str(e)
        }), 500

# ---- WebSocket Event Handlers ----

@socketio.on("connect", namespace="/market")
def on_connect():
    """Handle client connection."""
    session_id = request.sid
```

```
    user_agent = request.headers.get("User-Agent", "unknown")

    conn_mgr.add_connection(session_id, {"user_agent": user_agent})

    logger.info(f"Client connected: {session_id}")

    # Send welcome message with available symbols
    emit("connected", {
        "status": "connected",
        "session_id": session_id,
        "timestamp": time.time(),
        "symbols": engine.get_symbols()
    })

@socketio.on("disconnect", namespace="/market")
def on_disconnect():
    """Handle client disconnection."""
    session_id = request.sid
    conn_mgr.remove_connection(session_id)
    logger.info(f"Client disconnected: {session_id}")

@socketio.on("subscribe", namespace="/market")
def on_subscribe(data):
    """Handle subscription requests."""
    session_id = request.sid

    try:
        if not isinstance(data, dict):
            emit("error", {"message": "Invalid subscription data"})
            return

        subscription_type = data.get("type")
        symbol = data.get("symbol")

        if subscription_type == "l2_updates" and symbol:
            conn_mgr.subscribe_to_symbol(session_id, symbol.upper())

            # Send current snapshot
            snapshot = engine.get_book_snapshot(symbol, levels=10)
            emit("l2_update", snapshot)

            emit("subscribed", {
                "type": "l2_updates",
                "symbol": symbol.upper(),
                "timestamp": time.time()
            })

        elif subscription_type == "trades":
            conn_mgr.subscribe_to_trades(session_id)
```

```
            emit("subscribed", {
                "type": "trades",
                "timestamp": time.time()
            })

        else:
            emit("error", {"message": "Invalid subscription type or
missing symbol"})

    except Exception as e:
        logger.exception("Error in subscription")
        emit("error", {"message": f"Subscription failed: {str(e)}"})

@socketio.on("unsubscribe", namespace="/market")
def on_unsubscribe(data):
    """Handle unsubscription requests."""
    session_id = request.sid

    try:
        if not isinstance(data, dict):
            emit("error", {"message": "Invalid unsubscription data"})
            return

        subscription_type = data.get("type")
        symbol = data.get("symbol")

        if subscription_type == "l2_updates" and symbol:
            conn_mgr.unsubscribe_from_symbol(session_id,
symbol.upper())

            emit("unsubscribed", {
                "type": "l2_updates",
                "symbol": symbol.upper(),
                "timestamp": time.time()
            })

        elif subscription_type == "trades":
            conn_mgr.trade_subscribers.discard(session_id)

            emit("unsubscribed", {
                "type": "trades",
                "timestamp": time.time()
            })

        else:
            emit("error", {"message": "Invalid subscription type"})

    except Exception as e:
```

```
        logger.exception("Error in unsubscription")
        emit("error", {"message": f"Unsubscription failed: {str(e)}"})

@socketio.on("ping", namespace="/market")
def on_ping():
    """Handle ping requests."""
    emit("pong", {"timestamp": time.time()})

# ---- Error Handlers ----

@app.errorhandler(404)
def not_found(error):
    return jsonify({
        "error": "Endpoint not found",
        "error_code": "NOT_FOUND"
    }), 404

@app.errorhandler(405)
def method_not_allowed(error):
    return jsonify({
        "error": "Method not allowed",
        "error_code": "METHOD_NOT_ALLOWED"
    }), 405

@app.errorhandler(500)
def internal_error(error):
    logger.exception("Internal server error")
    return jsonify({
        "error": "Internal server error",
        "error_code": "INTERNAL_ERROR"
    }), 500

# ---- Background Tasks ----

def periodic_cleanup():
    """Periodic cleanup of stale connections and data."""
    while True:
        try:
            # This would run cleanup tasks
            eventlet.sleep(300)  # Sleep for 5 minutes

            # Clean up stale connections, expired rate limits, etc.
            logger.debug("Running periodic cleanup")

        except Exception as e:
            logger.exception("Error in periodic cleanup")
            eventlet.sleep(60)

# Start background tasks
eventlet.spawn(periodic_cleanup)
```

```
# ---- Application Factory ----

def create_app(config=None):
    """Application factory pattern."""
    if config:
        app.config.update(config)

    return app

# ---- Run Server ----

def run(host="0.0.0.0", port=5000, debug=False):
    """Run the API server."""
    logger.info(f"Starting Matching Engine API server on {host}:
{port}")
    logger.info(f"Supported symbols: {engine.get_symbols()}")
    logger.info(f"Debug mode: {debug}")

    socketio.run(
        app,
        host=host,
        port=port,
        debug=debug,
        use_reloader=False  # Disable reloader to prevent duplicate
processes
    )

if __name__ == "__main__":
    import argparse

    parser = argparse.ArgumentParser(description="Cryptocurrency
Matching Engine API")
    parser.add_argument("--host", default="0.0.0.0", help="Host to
bind to")
    parser.add_argument("--port", type=int, default=5000, help="Port
to bind to")
    parser.add_argument("--debug", action="store_true", help="Enable
debug mode")

    args = parser.parse_args()

    run(host=args.host, port=args.port, debug=args.debug)
```

engine/**init**.py

```
# engine/__init__.py
"""
Enhanced matching engine core package.
```

```
This package contains the core matching engine components with
enterprise-grade
features including:

- Thread-safe order book with price-time priority matching
- Support for multiple order types (Market, Limit, IOC, FOK)
- REG NMS-inspired trade-through protection
- High-performance data structures with O(log n) operations
- Comprehensive validation and error handling
- Fee calculation and trade reporting
- Event-driven architecture with callbacks

Components:
    - Order: Enhanced order representation with validation
    - OrderBook: Thread-safe order book with advanced matching
    - MatchingEngine: High-level engine coordinating multiple books
    - Trade: Trade execution records with fee calculations

Usage:
    from engine.matcher import MatchingEngine, create_crypto_engine
    from engine.order import Order, create_market_order,
create_limit_order
    from engine.book import OrderBook

    # Create engine with crypto defaults
    engine = create_crypto_engine()

    # Submit orders
    result = engine.submit_order("BTC-USDT", "buy", "limit", "0.001",
"30000")
"""

# Import only the classes that actually exist in the files
from .order import Order, create_market_order, create_limit_order
from .book import OrderBook, PriceLevel, Trade
from .matcher import (
    MatchingEngine,
    MatchingEngineError,
    OrderValidationError,
    InsufficientLiquidityError,
    SymbolNotFoundError,
    FeeCalculator,
    SymbolConfig,
    create_crypto_engine,
    create_forex_engine
)

__version__ = "2.0.0"
__all__ = [
    # Order components
```

```
    "Order", "create_market_order", "create_limit_order",

    # Book components
    "OrderBook", "PriceLevel", "Trade",

    # Engine components
    "MatchingEngine", "MatchingEngineError", "OrderValidationError",
    "InsufficientLiquidityError", "SymbolNotFoundError",
    "FeeCalculator", "SymbolConfig",

    # Factory functions
    "create_crypto_engine", "create_forex_engine"
]
```

engine/book.py

```
# engine/book.py
from collections import deque
from decimal import Decimal, getcontext
from typing import Dict, List, Optional, Tuple, Set
from engine.order import Order
import uuid
import time
import threading
import logging
from sortedcontainers import SortedDict
import bisect

# Set Decimal precision for financial calculations
getcontext().prec = 28

logger = logging.getLogger(__name__)


class PriceLevel:
    """
    Enhanced price level with better order management and thread
safety.
    """
    def __init__(self, price: Decimal):
        self.price: Decimal = price
        self.orders: deque[Order] = deque()
        self.aggregate: Decimal = Decimal("0")
        self.order_map: Dict[str, Order] = {}  # For O(1) order lookup
        self._lock = threading.RLock()

    def add_order(self, order: Order) -> None:
        """Thread-safe order addition."""
        with self._lock:
```

```
            self.orders.append(order)
            self.order_map[order.order_id] = order
            self.aggregate += order.remaining

    def peek_oldest(self) -> Optional[Order]:
        """Get the oldest order without removing it."""
        with self._lock:
            return self.orders[0] if self.orders else None

    def pop_oldest(self) -> Optional[Order]:
        """Remove and return the oldest order."""
        with self._lock:
            if not self.orders:
                return None
            order = self.orders.popleft()
            self.order_map.pop(order.order_id, None)
            self.aggregate -= order.remaining
            return order

    def remove_order(self, order_id: str) -> bool:
        """
        Remove a specific order by ID with O(1) lookup.
        Returns True if removed, False if not found.
        """
        with self._lock:
            if order_id not in self.order_map:
                return False

            order_to_remove = self.order_map.pop(order_id)

            # Remove from deque (this is O(n) but unavoidable)
            new_orders = deque()
            for order in self.orders:
                if order.order_id != order_id:
                    new_orders.append(order)

            self.orders = new_orders
            self.aggregate -= order_to_remove.remaining
            return True

    def update_order_fill(self, order_id: str, filled_qty: Decimal) ->
bool:
        """Update aggregate when an order is partially filled."""
        with self._lock:
            if order_id in self.order_map:
                self.aggregate -= filled_qty
                return True
            return False

    def is_empty(self) -> bool:
```

```
        """Check if price level is empty."""
        with self._lock:
            return len(self.orders) == 0 or self.aggregate <= 0

    def __len__(self) -> int:
        return len(self.orders)


class Trade:
    """Enhanced trade representation."""
    def __init__(self, symbol: str, price: Decimal, quantity: Decimal,

                 maker_order: Order, taker_order: Order, trade_seq:
int):
        self.trade_id = f"{symbol}-{trade_seq}-{str(uuid.uuid4())
[:8]}"
        self.timestamp = time.time()
        self.symbol = symbol
        self.price = price
        self.quantity = quantity
        self.maker_order_id = maker_order.order_id
        self.taker_order_id = taker_order.order_id
        self.aggressor_side = taker_order.side
        self.trade_seq = trade_seq

    def to_dict(self) -> dict:
        """Convert trade to dictionary for API response."""
        return {
            "timestamp": self.timestamp,
            "symbol": self.symbol,
            "trade_id": self.trade_id,
            "price": str(self.price),
            "quantity": str(self.quantity),
            "aggressor_side": self.aggressor_side,
            "maker_order_id": self.maker_order_id,
            "taker_order_id": self.taker_order_id
        }


class OrderBook:
    """
    Enhanced OrderBook with thread safety, better performance, and
comprehensive error handling.
    """
    def __init__(self, symbol: str, tick_size: Decimal =
Decimal("0.01"),
                 min_quantity: Decimal = Decimal("0.00000001")):
        self.symbol: str = symbol
        self.tick_size: Decimal = tick_size
        self.min_quantity: Decimal = min_quantity
```

```
        # Use SortedDict for O(log n) insertions and efficient range
queries
        # Keys are negative for bids to maintain descending order
        self.bids: SortedDict[Decimal, PriceLevel] = SortedDict()
        self.asks: SortedDict[Decimal, PriceLevel] = SortedDict()

        # Order tracking
        self.orders: Dict[str, Tuple[Order, PriceLevel]] = {}
        self.client_order_map: Dict[str, str] = {}  # client_order_id
-> order_id

        # Trade tracking
        self.trade_seq: int = 0

        # Thread safety
        self._lock = threading.RLock()

        # Statistics
        self.stats = {
            "total_orders": 0,
            "total_trades": 0,
            "total_volume": Decimal("0"),
            "last_trade_price": None,
            "last_trade_time": None
        }

    def _validate_price(self, price: Decimal) -> Decimal:
        """Validate and normalize price to tick size."""
        if price <= 0:
            raise ValueError("Price must be positive")

        # Round to nearest tick
        ticks = price / self.tick_size
        rounded_ticks = round(ticks)
        return rounded_ticks * self.tick_size

    def _validate_quantity(self, quantity: Decimal) -> None:
        """Validate quantity."""
        if quantity <= 0:
            raise ValueError("Quantity must be positive")
        if quantity < self.min_quantity:
            raise ValueError(f"Quantity below minimum:
{self.min_quantity}")

    def best_bid(self) -> Optional[Decimal]:
        """Get best bid price (highest)."""
        with self._lock:
            return -self.bids.peekitem(-1)[0] if self.bids else None
```

```
    def best_ask(self) -> Optional[Decimal]:
        """Get best ask price (lowest)."""
        with self._lock:
            return self.asks.peekitem(0)[0] if self.asks else None

    def spread(self) -> Optional[Decimal]:
        """Calculate bid-ask spread."""
        bb, ba = self.best_bid(), self.best_ask()
        return ba - bb if bb and ba else None

    def mid_price(self) -> Optional[Decimal]:
        """Calculate mid price."""
        bb, ba = self.best_bid(), self.best_ask()
        return (bb + ba) / 2 if bb and ba else None

    def get_bbo(self) -> Dict:
        """Get Best Bid Offer snapshot."""
        with self._lock:
            bb = self.best_bid()
            ba = self.best_ask()
            return {
                "symbol": self.symbol,
                "best_bid": str(bb) if bb else None,
                "best_ask": str(ba) if ba else None,
                "spread": str(self.spread()) if self.spread() else
None,
                "mid_price": str(self.mid_price()) if self.mid_price()
else None,
                "timestamp": time.time()
            }

    def get_depth(self, levels: int = 10) -> Dict:
        """Get order book depth (L2 data)."""
        with self._lock:
            bids = []
            asks = []

            # Get top N bid levels (highest prices first)
            # Need to reverse the items manually since SortedDict
doesn't support reverse parameter
            bid_items = list(self.bids.items())
            bid_items.reverse()  # Reverse to get highest prices first

            for neg_price, level in bid_items:
                if len(bids) >= levels:
                    break
                if level.aggregate > 0:  # Only show levels with
quantity
                    price = -neg_price
                    bids.append([str(price), str(level.aggregate)])
```

```
            # Get top N ask levels (lowest prices first)
            for price, level in self.asks.items():
                if len(asks) >= levels:
                    break
                if level.aggregate > 0:  # Only show levels with
quantity
                    asks.append([str(price), str(level.aggregate)])

            return {
                "timestamp": time.time(),
                "symbol": self.symbol,
                "bids": bids,
                "asks": asks
            }

    def _add_resting_order(self, order: Order) -> None:
        """Add a resting limit order to the book."""
        price = self._validate_price(order.price)

        if order.side == "buy":
            neg_price = -price  # Store as negative for descending
sort
            if neg_price not in self.bids:
                self.bids[neg_price] = PriceLevel(price)
            level = self.bids[neg_price]
        else:
            if price not in self.asks:
                self.asks[price] = PriceLevel(price)
            level = self.asks[price]

        level.add_order(order)
        self.orders[order.order_id] = (order, level)

        if order.client_order_id:
            self.client_order_map[order.client_order_id] =
order.order_id

    def _remove_empty_level(self, price: Decimal, side: str) -> None:
        """Remove empty price level."""
        if side == "buy":
            neg_price = -price
            if neg_price in self.bids and
self.bids[neg_price].is_empty():
                del self.bids[neg_price]
        else:
            if price in self.asks and self.asks[price].is_empty():
                del self.asks[price]

    def cancel_order(self, order_id: str = None, client_order_id: str
```

```
= None) -> bool:
        """
        Cancel order by order_id or client_order_id.
        Returns True if cancelled, False if not found.
        """
        with self._lock:
            # Resolve client_order_id to order_id
            if client_order_id and not order_id:
                order_id = self.client_order_map.get(client_order_id)

            if not order_id or order_id not in self.orders:
                return False

            order, level = self.orders.pop(order_id)

            # Remove from client mapping if exists
            if order.client_order_id:
                self.client_order_map.pop(order.client_order_id, None)

            # Cancel the order
            order.cancel()

            # Remove from price level
            success = level.remove_order(order_id)

            # Clean up empty level
            if level.is_empty():
                self._remove_empty_level(level.price, order.side)

            return success

    def _calculate_available_liquidity(self, incoming_side: str,
max_price: Decimal = None) -> Decimal:
        """
        Calculate total available liquidity for matching.
        Improved FOK calculation that considers all marketable levels.
        """
        total = Decimal("0")

        if incoming_side == "buy":
            # Buyer takes asks from lowest price upward
            for price, level in self.asks.items():
                if max_price and price > max_price:
                    break
                total += level.aggregate
        else:
            # Seller takes bids from highest price downward
            for neg_price, level in self.bids.items(reverse=True):
                price = -neg_price
```

```
                if max_price and price < max_price:
                    break
                total += level.aggregate

        return total

    def _is_order_marketable(self, order: Order) -> bool:
        """Check if order can be immediately matched."""
        if order.is_market():
            return True

        if not order.price:
            return False

        if order.side == "buy":
            best_ask = self.best_ask()
            return best_ask is not None and order.price >= best_ask
        else:
            best_bid = self.best_bid()
            return best_bid is not None and order.price <= best_bid

    def _execute_trade(self, maker_order: Order, taker_order: Order,
                       quantity: Decimal, price: Decimal) -> Trade:
        """Execute a trade between two orders."""
        # Update order quantities
        maker_order.fill(quantity)
        taker_order.fill(quantity)

        # Update price level aggregate
        if maker_order.order_id in self.orders:
            _, level = self.orders[maker_order.order_id]
            level.update_order_fill(maker_order.order_id, quantity)

        # Create trade record
        self.trade_seq += 1
        trade = Trade(
            symbol=self.symbol,
            price=price,
            quantity=quantity,
            maker_order=maker_order,
            taker_order=taker_order,
            trade_seq=self.trade_seq
        )

        # Update statistics
        self.stats["total_trades"] += 1
        self.stats["total_volume"] += quantity
        self.stats["last_trade_price"] = price
        self.stats["last_trade_time"] = trade.timestamp
```

```
        return trade

    def _match_order(self, incoming_order: Order) -> List[Trade]:
        """
        Core matching logic with improved error handling and FOK
support.
        """
        trades: List[Trade] = []

        # Choose opposite side of book
        if incoming_order.side == "buy":
            opposite_levels = self.asks
            price_check = lambda p: incoming_order.is_market() or
incoming_order.price >= p
        else:
            opposite_levels = self.bids
            price_check = lambda p: incoming_order.is_market() or
incoming_order.price <= (-p if incoming_order.side == "sell" else p)

        # FOK pre-validation: ensure sufficient liquidity
        if incoming_order.is_fok():
            max_price = incoming_order.price if not
incoming_order.is_market() else None
            available =
self._calculate_available_liquidity(incoming_order.side, max_price)
            if available < incoming_order.remaining:
                logger.info(f"FOK order {incoming_order.order_id}
rejected: insufficient liquidity")
                incoming_order.reject("Insufficient liquidity")
                return []

        # Process opposite levels in price-time priority
        levels_to_remove = []

        if incoming_order.side == "buy":
            # Take asks from lowest to highest (natural order)
            level_items = list(opposite_levels.items())
        else:
            # Take bids from highest to lowest (reverse order)
            level_items = list(opposite_levels.items())
            level_items.reverse()

        for key, level in level_items:
            if incoming_order.remaining <= 0:
                break

            # Get the actual price for this level
            if incoming_order.side == "buy":
```

```
                level_price = key  # For asks, key is the actual price
            else:
                level_price = -key  # For bids, key is negative, so
convert back

            # Check if this price level is marketable
            if incoming_order.side == "buy":
                # For buy orders, check against ask price
                if not price_check(level_price):
                    break
            else:
                # For sell orders, check against bid price (key is
negative)
                if not price_check(key):
                    break

            # Match against orders at this price level
            while incoming_order.remaining > 0 and not
level.is_empty():
                maker_order = level.peek_oldest()
                if not maker_order or not maker_order.is_active:
                    level.pop_oldest()
                    continue

                # Calculate trade quantity
                trade_qty = min(maker_order.remaining,
incoming_order.remaining)

                # Execute trade at the resting order's price (price
improvement for aggressor)
                trade = self._execute_trade(maker_order,
incoming_order, trade_qty, level_price)
                trades.append(trade)

                # Remove maker order if fully filled
                if maker_order.remaining <= 0:
                    filled_order = level.pop_oldest()
                    self.orders.pop(maker_order.order_id, None)
                    if maker_order.client_order_id:

self.client_order_map.pop(maker_order.client_order_id, None)

            # Mark empty levels for removal
            if level.is_empty():
                levels_to_remove.append((key, level_price,
incoming_order.side))

        # Clean up empty levels
        for key, price, side in levels_to_remove:
```

```
            opposite_side = "sell" if side == "buy" else "buy"
            self._remove_empty_level(price, opposite_side)

        return trades

    def submit_order(self, order: Order) -> Tuple[List[Trade], bool]:
        """
        Submit order to the book with comprehensive validation and
matching.

        Returns:
            Tuple[List[Trade], bool]: (trades_executed, order_resting)
        """
        with self._lock:
            try:
                # Validate order
                if order.price:
                    self._validate_price(order.price)
                self._validate_quantity(order.remaining)

                # Update statistics
                self.stats["total_orders"] += 1

                trades = []
                order_resting = False

                # Attempt matching if marketable
                if self._is_order_marketable(order):
                    trades = self._match_order(order)

                # Handle remaining quantity based on order type
                if order.remaining > 0 and order.is_active:
                    if order.order_type == "market":
                        # Market order remainder cancelled (no more
liquidity)
                        order.cancel()
                        logger.warning(f"Market order {order.order_id}
partially cancelled: no liquidity")

                    elif order.order_type == "ioc":
                        # IOC remainder cancelled
                        order.cancel()

                    elif order.order_type == "fok":
                        # FOK should not reach here due to pre-check,
but safety cancel
                        order.cancel()
                        trades = []  # Cancel all trades for FOK
                        logger.error(f"FOK order {order.order_id}
inconsistent state")
```

```
                    elif order.order_type == "limit":
                        # Rest limit order on book
                        self._add_resting_order(order)
                        order_resting = True

                return trades, order_resting

            except Exception as e:
                logger.error(f"Error submitting order
{order.order_id}: {e}")
                order.reject(str(e))
                raise

    def get_order_status(self, order_id: str = None, client_order_id:
str = None) -> Optional[Dict]:
        """Get current status of an order."""
        with self._lock:
            # Resolve client_order_id
            if client_order_id and not order_id:
                order_id = self.client_order_map.get(client_order_id)

            if not order_id or order_id not in self.orders:
                return None

            order, _ = self.orders[order_id]
            return order.to_dict()

    def get_statistics(self) -> Dict:
        """Get book statistics."""
        with self._lock:
            stats = self.stats.copy()
            stats.update({
                "active_orders": len(self.orders),
                "bid_levels": len(self.bids),
                "ask_levels": len(self.asks),
                "total_volume": str(stats["total_volume"]),
                "last_trade_price": str(stats["last_trade_price"]) if
stats["last_trade_price"] else None
            })
            return stats

    # Legacy compatibility methods
    def top_n(self, n: int = 10) -> Dict:
        """Legacy method - use get_depth() instead."""
        return self.get_depth(n)

    def match(self, incoming: Order) -> List[Dict]:
        """Legacy method - use submit_order() instead."""
        trades, _ = self.submit_order(incoming)
```

```
        return [trade.to_dict() for trade in trades]

    def submit(self, order: Order) -> List[Dict]:
        """Legacy method - use submit_order() instead."""
        trades, _ = self.submit_order(order)
        return [trade.to_dict() for trade in trades]
```

engine/matcher.py

```
# engine/matcher.py
from decimal import Decimal, InvalidOperation
from engine.book import OrderBook, Trade
from engine.order import Order
from typing import Dict, Any, Optional, List, Callable
import threading
import logging
import time
from collections import defaultdict


logger = logging.getLogger(__name__)


class MatchingEngineError(Exception):
    """Base exception for matching engine errors."""
    pass


class OrderValidationError(MatchingEngineError):
    """Raised when order validation fails."""
    pass


class InsufficientLiquidityError(MatchingEngineError):
    """Raised when there's insufficient liquidity for FOK orders."""
    pass


class SymbolNotFoundError(MatchingEngineError):
    """Raised when symbol is not supported."""
    pass


class FeeCalculator:
    """Simple maker-taker fee calculator."""

    def __init__(self, maker_fee: Decimal = Decimal("0.001"),
taker_fee: Decimal = Decimal("0.001")):
```

```
        self.maker_fee = maker_fee  # 0.1% default
        self.taker_fee = taker_fee  # 0.1% default

    def calculate_maker_fee(self, trade_value: Decimal) -> Decimal:
        """Calculate maker fee for a trade."""
        return trade_value * self.maker_fee

    def calculate_taker_fee(self, trade_value: Decimal) -> Decimal:
        """Calculate taker fee for a trade."""
        return trade_value * self.taker_fee

    def calculate_fees(self, trade: Trade) -> Dict[str, Decimal]:
        """Calculate both maker and taker fees for a trade."""
        trade_value = trade.price * trade.quantity
        return {
            "maker_fee": self.calculate_maker_fee(trade_value),
            "taker_fee": self.calculate_taker_fee(trade_value),
            "trade_value": trade_value
        }


class SymbolConfig:
    """Configuration for a trading symbol."""

    def __init__(self, symbol: str, tick_size: Decimal =
Decimal("0.01"),
                 min_quantity: Decimal = Decimal("0.00000001"),
                 max_quantity: Decimal = Decimal("1000000"),
                 min_notional: Decimal = Decimal("10")):
        self.symbol = symbol
        self.tick_size = tick_size
        self.min_quantity = min_quantity
        self.max_quantity = max_quantity
        self.min_notional = min_notional  # Minimum order value


class MatchingEngine:
    """
    High-performance matching engine optimized for low latency and
high throughput.
    Simplified design focused on core functionality.
    """

    def __init__(self, fee_calculator: FeeCalculator = None):
        # Order books by symbol
        self.books: Dict[str, OrderBook] = {}

        # Symbol configurations
        self.symbol_configs: Dict[str, SymbolConfig] = {}
```

```
        # Fee calculator
        self.fee_calculator = fee_calculator or FeeCalculator()

        # Thread safety - single lock for simplicity and correctness
        self._lock = threading.RLock()

        # Event handlers - simple synchronous callbacks
        self.trade_handlers: List[Callable[[Trade], None]] = []
        self.order_handlers: List[Callable[[Order, str], None]] = []

        # Performance metrics
        self.metrics = {
            "orders_processed": 0,
            "trades_executed": 0,
            "total_volume": Decimal("0"),
            "avg_latency_ms": 0.0,
            "start_time": time.time()
        }

        # Error tracking
        self.error_counts = defaultdict(int)

        logger.info("MatchingEngine initialized")

    def add_symbol(self, symbol: str, config: SymbolConfig = None) ->
None:
        """Add a new trading symbol with optional configuration."""
        with self._lock:
            if symbol in self.books:
                logger.warning(f"Symbol {symbol} already exists")
                return

            if not config:
                config = SymbolConfig(symbol)

            self.symbol_configs[symbol] = config
            self.books[symbol] = OrderBook(
                symbol=symbol,
                tick_size=config.tick_size,
                min_quantity=config.min_quantity
            )

            logger.info(f"Added symbol {symbol} with
tick_size={config.tick_size}")

    def remove_symbol(self, symbol: str) -> bool:
        """Remove a trading symbol. Returns True if removed."""
        with self._lock:
            if symbol not in self.books:
```

```
                return False

            # Check for active orders
            book = self.books[symbol]
            if len(book.orders) > 0:
                logger.warning(f"Cannot remove symbol {symbol}: has
active orders")
                return False

            del self.books[symbol]
            del self.symbol_configs[symbol]
            logger.info(f"Removed symbol {symbol}")
            return True

    def get_symbols(self) -> List[str]:
        """Get list of supported symbols."""
        with self._lock:
            return list(self.books.keys())

    def _get_book(self, symbol: str) -> OrderBook:
        """Get order book for symbol, creating if necessary."""
        if symbol not in self.books:
            # Auto-create with default config
            self.add_symbol(symbol)
        return self.books[symbol]

    def _validate_order_params(self, symbol: str, side: str,
order_type: str,
                               quantity, price: Optional[float] = None)
-> tuple:
        """Validate and normalize order parameters."""
        # Normalize strings
        symbol = symbol.upper().strip()
        side = side.lower().strip()
        order_type = order_type.lower().strip()

        # Validate basic params
        if not symbol:
            raise OrderValidationError("Symbol cannot be empty")

        if side not in ("buy", "sell"):
            raise OrderValidationError("Side must be 'buy' or 'sell'")

        if order_type not in ("market", "limit", "ioc", "fok"):
            raise OrderValidationError("Invalid order type")

        # Convert and validate quantity
        try:
            quantity = Decimal(str(quantity))
            if quantity <= 0:
```

```
                raise OrderValidationError("Quantity must be
positive")
        except (InvalidOperation, TypeError, ValueError) as e:
            raise OrderValidationError(f"Invalid quantity: {e}")

        # Convert and validate price
        validated_price = None
        if price is not None:
            try:
                validated_price = Decimal(str(price))
                if validated_price <= 0:
                    raise OrderValidationError("Price must be
positive")
            except (InvalidOperation, TypeError, ValueError) as e:
                raise OrderValidationError(f"Invalid price: {e}")

        # Check symbol-specific limits
        if symbol in self.symbol_configs:
            config = self.symbol_configs[symbol]

            if quantity < config.min_quantity:
                raise OrderValidationError(f"Quantity below minimum:
{config.min_quantity}")

            if quantity > config.max_quantity:
                raise OrderValidationError(f"Quantity above maximum:
{config.max_quantity}")

            if validated_price and config.min_notional:
                notional = validated_price * quantity
                if notional < config.min_notional:
                    raise OrderValidationError(f"Order value below
minimum: {config.min_notional}")

        return symbol, side, order_type, quantity, validated_price

    def add_trade_handler(self, handler: Callable[[Trade], None]) ->
None:
        """Add a trade event handler."""
        self.trade_handlers.append(handler)

    def add_order_handler(self, handler: Callable[[Order, str], None])
-> None:
        """Add an order event handler."""
        self.order_handlers.append(handler)

    def _emit_trade_event(self, trade: Trade) -> None:
        """Emit trade event to all handlers."""
        for handler in self.trade_handlers:
            try:
```

```
                handler(trade)
            except Exception as e:
                logger.error(f"Trade handler error: {e}")

    def _emit_order_event(self, order: Order, event_type: str) ->
None:
        """Emit order event to all handlers."""
        for handler in self.order_handlers:
            try:
                handler(order, event_type)
            except Exception as e:
                logger.error(f"Order handler error: {e}")

    def submit_order(self, symbol: str, side: str, order_type: str,
                     quantity, price: Optional[float] = None,
                     client_order_id: str = None) -> Dict[str, Any]:
        """
        Submit an order with comprehensive validation and error
handling.

        Returns:
            Dict containing order_id, trades, fees, and market data
        """
        start_time = time.time()

        try:
            # Validate parameters
            symbol, side, order_type, quantity, validated_price =
self._validate_order_params(
                symbol, side, order_type, quantity, price
            )

            # Create order
            order = Order(
                symbol=symbol,
                side=side,
                order_type=order_type,
                quantity=quantity,
                price=validated_price,
                client_order_id=client_order_id
            )

            # Process order with single lock
            with self._lock:
                # Get book and submit order
                book = self._get_book(symbol)
                trades, order_resting = book.submit_order(order)

                # Calculate fees for trades
```

```
                trade_dicts = []
                total_fees = {"maker_fees": Decimal("0"),
"taker_fees": Decimal("0")}

                for trade in trades:
                    trade_dict = trade.to_dict()

                    # Add fee information
                    fees = self.fee_calculator.calculate_fees(trade)
                    trade_dict.update({
                        "maker_fee": str(fees["maker_fee"]),
                        "taker_fee": str(fees["taker_fee"]),
                        "trade_value": str(fees["trade_value"])
                    })

                    total_fees["maker_fees"] += fees["maker_fee"]
                    total_fees["taker_fees"] += fees["taker_fee"]

                    trade_dicts.append(trade_dict)

                    # Emit trade event
                    self._emit_trade_event(trade)

                # Emit order events
                if trades:
                    if order.remaining <= 0:
                        self._emit_order_event(order, "filled")
                    else:
                        self._emit_order_event(order,
"partially_filled")

                if order_resting:
                    self._emit_order_event(order, "resting")

                if order.status == "cancelled":
                    self._emit_order_event(order, "cancelled")

                if order.status == "rejected":
                    self._emit_order_event(order, "rejected")

                # Update metrics
                self.metrics["orders_processed"] += 1
                self.metrics["trades_executed"] += len(trades)
                if trades:
                    for trade in trades:
                        self.metrics["total_volume"] += trade.quantity

                # Calculate latency
                latency_ms = (time.time() - start_time) * 1000
```

```
                self.metrics["avg_latency_ms"] = (
                    (self.metrics["avg_latency_ms"] *
(self.metrics["orders_processed"] - 1) + latency_ms) /
                    self.metrics["orders_processed"]
                )

                # Build response
                response = {
                    "order_id": order.order_id,
                    "client_order_id": order.client_order_id,
                    "status": order.status,
                    "filled_quantity": str(order.filled),
                    "remaining_quantity": str(order.remaining),
                    "trades": trade_dicts,
                    "total_maker_fees": str(total_fees["maker_fees"]),
                    "total_taker_fees": str(total_fees["taker_fees"]),
                    "bbo": book.get_bbo(),
                    "processing_time_ms": latency_ms
                }

                logger.debug(f"Order submitted: {order.order_id}
{side} {quantity}@{validated_price or 'MKT'} -> {len(trades)} trades")
                return response

        except Exception as e:
            self.error_counts[type(e).__name__] += 1
            logger.error(f"Order submission failed: {e}")

            error_response = {
                "error": str(e),
                "error_type": type(e).__name__,
                "processing_time_ms": (time.time() - start_time) *
1000
            }

            if isinstance(e, (OrderValidationError,
InsufficientLiquidityError)):
                error_response["error_code"] = "VALIDATION_ERROR"
            else:
                error_response["error_code"] = "INTERNAL_ERROR"

            return error_response

    def cancel_order(self, symbol: str, order_id: str = None,
                     client_order_id: str = None) -> Dict[str, Any]:
        """Cancel an order by order_id or client_order_id."""
        start_time = time.time()

        try:
```

```
            if not order_id and not client_order_id:
                raise OrderValidationError("Must provide order_id or
client_order_id")

            symbol = symbol.upper().strip()

            with self._lock:
                book = self._get_book(symbol)

                # Get order status before cancellation
                order_status = book.get_order_status(order_id,
client_order_id)
                if not order_status:
                    return {
                        "success": False,
                        "error": "Order not found",
                        "processing_time_ms": (time.time() -
start_time) * 1000
                    }

                # Cancel the order
                success = book.cancel_order(order_id, client_order_id)

                if success:
                    # Create order object for event
                    cancelled_order = Order(
                        symbol=symbol,
                        side=order_status["side"],
                        order_type=order_status["order_type"],
                        quantity=Decimal(order_status["quantity"]),
                        price=Decimal(order_status["price"]) if
order_status["price"] else None
                    )
                    cancelled_order.order_id =
order_status["order_id"]
                    cancelled_order.status = "cancelled"

                    self._emit_order_event(cancelled_order,
"cancelled")
                    logger.debug(f"Order cancelled:
{order_status['order_id']}")

                return {
                    "success": success,
                    "order_id": order_status["order_id"],
                    "client_order_id":
order_status.get("client_order_id"),
                    "bbo": book.get_bbo(),
                    "processing_time_ms": (time.time() - start_time) *
```

```
1000
                }

        except Exception as e:
            self.error_counts[type(e).__name__] += 1
            logger.error(f"Order cancellation failed: {e}")
            return {
                "success": False,
                "error": str(e),
                "processing_time_ms": (time.time() - start_time) *
1000
            }

    def get_order_status(self, symbol: str, order_id: str = None,
                         client_order_id: str = None) ->
Optional[Dict]:
        """Get order status."""
        try:
            symbol = symbol.upper().strip()
            with self._lock:
                if symbol not in self.books:
                    return None

                book = self.books[symbol]
                return book.get_order_status(order_id,
client_order_id)

        except Exception as e:
            logger.error(f"Error getting order status: {e}")
            return None

    def get_book_snapshot(self, symbol: str, levels: int = 10) ->
Dict:
        """Get L2 order book snapshot."""
        try:
            symbol = symbol.upper().strip()
            with self._lock:
                book = self._get_book(symbol)
                return book.get_depth(levels)

        except Exception as e:
            logger.error(f"Error getting book snapshot: {e}")
            return {
                "timestamp": time.time(),
                "symbol": symbol,
                "error": str(e),
                "bids": [],
                "asks": []
            }
```

```
    def get_bbo(self, symbol: str) -> Dict:
        """Get Best Bid Offer for symbol."""
        try:
            symbol = symbol.upper().strip()
            with self._lock:
                book = self._get_book(symbol)
                return book.get_bbo()

        except Exception as e:
            logger.error(f"Error getting BBO: {e}")
            return {
                "symbol": symbol,
                "error": str(e),
                "best_bid": None,
                "best_ask": None,
                "timestamp": time.time()
            }

    def get_statistics(self, symbol: str = None) -> Dict:
        """Get engine or symbol-specific statistics."""
        try:
            with self._lock:
                if symbol:
                    symbol = symbol.upper().strip()
                    if symbol not in self.books:
                        return {"error": f"Symbol {symbol} not found"}
                    return self.books[symbol].get_statistics()
                else:
                    # Engine-wide statistics
                    uptime = time.time() - self.metrics["start_time"]
                    stats = self.metrics.copy()
                    stats.update({
                        "uptime_seconds": uptime,
                        "symbols_count": len(self.books),
                        "total_volume": str(stats["total_volume"]),
                        "orders_per_second": stats["orders_processed"]
/ max(uptime, 1),
                        "error_counts": dict(self.error_counts)
                    })
                    return stats

        except Exception as e:
            logger.error(f"Error getting statistics: {e}")
            return {"error": str(e)}

    def reset_statistics(self) -> None:
        """Reset performance metrics."""
        with self._lock:
            self.metrics = {
                "orders_processed": 0,
```

```
                "trades_executed": 0,
                "total_volume": Decimal("0"),
                "avg_latency_ms": 0.0,
                "start_time": time.time()
            }
            self.error_counts.clear()
            logger.info("Statistics reset")

    # Legacy compatibility methods
    def cancel(self, symbol: str, order_id: str) -> bool:
        """Legacy method - use cancel_order() instead."""
        result = self.cancel_order(symbol, order_id=order_id)
        return result.get("success", False)

    def top_n(self, symbol: str, n: int = 10) -> Dict:
        """Legacy method - use get_book_snapshot() instead."""
        return self.get_book_snapshot(symbol, n)


# Convenience factory functions
def create_crypto_engine() -> MatchingEngine:
    """Create a matching engine with common crypto symbol
configurations."""
    engine = MatchingEngine()

    # Add common crypto pairs
    crypto_symbols = {
        "BTC-USDT": SymbolConfig("BTC-USDT", Decimal("0.01"),
Decimal("0.00001")),
        "ETH-USDT": SymbolConfig("ETH-USDT", Decimal("0.01"),
Decimal("0.0001")),
        "BNB-USDT": SymbolConfig("BNB-USDT", Decimal("0.01"),
Decimal("0.001")),
        "SOL-USDT": SymbolConfig("SOL-USDT", Decimal("0.01"),
Decimal("0.01")),
    }

    for symbol, config in crypto_symbols.items():
        engine.add_symbol(symbol, config)

    return engine


def create_forex_engine() -> MatchingEngine:
    """Create a matching engine with common forex pair
configurations."""
    engine = MatchingEngine(FeeCalculator(Decimal("0.0001"),
Decimal("0.0001")))  # Lower fees for forex

    # Add major forex pairs
```

```
    forex_symbols = {
        "EUR-USD": SymbolConfig("EUR-USD", Decimal("0.00001"),
Decimal("1000")),
        "GBP-USD": SymbolConfig("GBP-USD", Decimal("0.00001"),
Decimal("1000")),
        "USD-JPY": SymbolConfig("USD-JPY", Decimal("0.001"),
Decimal("1000")),
    }

    for symbol, config in forex_symbols.items():
        engine.add_symbol(symbol, config)

    return engine
```

engine/order.py

```
# engine/order.py
from dataclasses import dataclass, field
from decimal import Decimal, InvalidOperation, getcontext
from uuid import uuid4
import time
from typing import Optional

# Set high precision for financial calculations
getcontext().prec = 28


@dataclass
class Order:
    """
    Enhanced Order dataclass with validation and better error
handling.

    Fields:
        symbol: trading pair (e.g., "BTC-USDT")
        side: "buy" or "sell"
        quantity: Decimal (initial quantity)
        price: Decimal | None (None for market orders)
        order_type: "market", "limit", "ioc", "fok"
        order_id: unique id string
        timestamp: float (epoch seconds)
        remaining: Decimal (remaining qty to fill)
        filled: Decimal (filled quantity)
        status: order status string
    """
    symbol: str
    side: str                    # "buy" or "sell"
    quantity: Decimal
    price: Optional[Decimal]     # None for market orders
```

```
    order_type: str              # "market", "limit", "ioc", "fok"
    order_id: str = field(default_factory=lambda: str(uuid4()))
    timestamp: float = field(default_factory=time.time)
    remaining: Optional[Decimal] = None
    filled: Decimal = field(default_factory=lambda: Decimal("0"))
    status: str = field(default="pending")
    client_order_id: Optional[str] = None

    def __post_init__(self):
        """Validate and normalize order data."""
        self._validate_and_normalize()
        if self.remaining is None:
            self.remaining = Decimal(self.quantity)

    def _validate_and_normalize(self):
        """Basic validation and type conversion."""
        # Normalize strings
        if not isinstance(self.symbol, str) or not
self.symbol.strip():
            raise ValueError("Symbol must be a non-empty string")
        self.symbol = self.symbol.upper().strip()

        # Validate side
        if isinstance(self.side, str):
            self.side = self.side.lower().strip()
        if self.side not in ("buy", "sell"):
            raise ValueError("Side must be 'buy' or 'sell'")

        # Validate order type
        if isinstance(self.order_type, str):
            self.order_type = self.order_type.lower().strip()
        if self.order_type not in ("market", "limit", "ioc", "fok"):
            raise ValueError("Invalid order type")

        # Convert and validate quantity
        try:
            if isinstance(self.quantity, (int, float, str)):
                self.quantity = Decimal(str(self.quantity))
            if self.quantity <= 0:
                raise ValueError("Quantity must be positive")
        except (InvalidOperation, TypeError) as e:
            raise ValueError(f"Invalid quantity: {e}")

        # Convert and validate price
        if self.price is not None:
            try:
                if isinstance(self.price, (int, float, str)):
                    self.price = Decimal(str(self.price))
                if self.price <= 0:
                    raise ValueError("Price must be positive")
```

```
            except (InvalidOperation, TypeError) as e:
                raise ValueError(f"Invalid price: {e}")

        # Validate price requirements
        if self.order_type in ("limit", "ioc", "fok") and self.price
is None:
            raise ValueError(f"{self.order_type} orders require a
price")

        if self.order_type == "market" and self.price is not None:
            raise ValueError("Market orders cannot have a price")

        # Convert remaining and filled if provided
        if self.remaining is not None:
            try:
                if isinstance(self.remaining, (int, float, str)):
                    self.remaining = Decimal(str(self.remaining))
            except (InvalidOperation, TypeError) as e:
                raise ValueError(f"Invalid remaining quantity: {e}")

        if isinstance(self.filled, (int, float, str)):
            self.filled = Decimal(str(self.filled))

    @property
    def is_active(self) -> bool:
        """Check if order is active (can be matched or cancelled)."""
        return self.status in ("pending", "partially_filled")

    def is_market(self) -> bool:
        """Check if this is a market order."""
        return self.order_type == "market"

    def is_limit(self) -> bool:
        """Check if this is a limit order."""
        return self.order_type == "limit"

    def is_ioc(self) -> bool:
        """Check if this is an IOC order."""
        return self.order_type == "ioc"

    def is_fok(self) -> bool:
        """Check if this is a FOK order."""
        return self.order_type == "fok"

    def fill(self, quantity: Decimal) -> None:
        """
        Fill part of the order.

        Args:
            quantity: Amount to fill
```

```
        """
        if quantity <= 0:
            raise ValueError("Fill quantity must be positive")

        if quantity > self.remaining:
            raise ValueError("Cannot fill more than remaining
quantity")

        self.filled += quantity
        self.remaining -= quantity

        # Update status
        if self.remaining <= 0:
            self.status = "filled"
        elif self.filled > 0:
            self.status = "partially_filled"

    def cancel(self) -> bool:
        """Cancel the order if it's active."""
        if not self.is_active:
            return False

        self.status = "cancelled"
        return True

    def reject(self, reason: str = None) -> None:
        """Reject the order."""
        self.status = "rejected"

    def to_dict(self) -> dict:
        """Convert order to dictionary for serialization."""
        return {
            "order_id": self.order_id,
            "client_order_id": self.client_order_id,
            "symbol": self.symbol,
            "side": self.side,
            "order_type": self.order_type,
            "quantity": str(self.quantity),
            "price": str(self.price) if self.price is not None else
None,
            "remaining": str(self.remaining),
            "filled": str(self.filled),
            "status": self.status,
            "timestamp": self.timestamp
        }

    def __repr__(self) -> str:
        price_str = str(self.price) if self.price is not None else
"MKT"
        return (f"<Order {self.order_id[:8]} {self.side.upper()} "
```

```
                f"{self.quantity}@{price_str} ({self.order_type})
{self.status}>")

    def __eq__(self, other) -> bool:
        """Orders are equal if they have the same order_id."""
        if not isinstance(other, Order):
            return False
        return self.order_id == other.order_id

    def __hash__(self) -> int:
        """Hash based on order_id for use in sets/dicts."""
        return hash(self.order_id)


# Convenience factory functions for common order types
def create_market_order(symbol: str, side: str, quantity,
client_order_id: str = None) -> Order:
    """Create a market order."""
    return Order(
        symbol=symbol,
        side=side,
        quantity=quantity,
        price=None,
        order_type="market",
        client_order_id=client_order_id
    )

def create_limit_order(symbol: str, side: str, quantity, price,
client_order_id: str = None) -> Order:
    """Create a limit order."""
    return Order(
        symbol=symbol,
        side=side,
        quantity=quantity,
        price=price,
        order_type="limit",
        client_order_id=client_order_id
    )
```

frontend/index.html

```
'''
<!DOCTYPE html>
<html lang="en">
```

```
<head>
    <meta charset="utf-8">
    <title>Cryptocurrency Matching Engine – Advanced Trading
Interface</title>
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <link
href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/6.4.0/css/
all.min.css" rel="stylesheet">
    <style>
        * {
            box-sizing: border-box;
            margin: 0;
            padding: 0;
        }

        body {
            font-family: -apple-system, BlinkMacSystemFont, 'Segoe
UI', Roboto, Oxygen, Ubuntu, Cantarell, sans-serif;
            background: linear-gradient(135deg, #667eea 0%, #764ba2
100%);
            color: #333;
            min-height: 100vh;
            font-size: 14px;
        }

        .header {
            background: rgba(255, 255, 255, 0.95);
            backdrop-filter: blur(10px);
            border-bottom: 1px solid rgba(0, 0, 0, 0.1);
            padding: 1rem 2rem;
            position: sticky;
            top: 0;
            z-index: 100;
            box-shadow: 0 2px 20px rgba(0, 0, 0, 0.1);
        }

        .header-content {
            max-width: 1400px;
            margin: 0 auto;
            display: flex;
            justify-content: space-between;
            align-items: center;
        }

        .logo {
            font-size: 1.5rem;
            font-weight: 700;
            color: #667eea;
            display: flex;
```

```
            align-items: center;
            gap: 0.5rem;
        }

        .status-bar {
            display: flex;
            gap: 2rem;
            align-items: center;
        }

        .status-item {
            display: flex;
            flex-direction: column;
            align-items: center;
            gap: 0.25rem;
        }

        .status-label {
            font-size: 0.75rem;
            color: #666;
            text-transform: uppercase;
            letter-spacing: 0.5px;
        }

        .status-value {
            font-weight: 600;
            font-size: 0.9rem;
        }

        .status-connected { color: #10b981; }
        .status-disconnected { color: #ef4444; }

        .container {
            max-width: 1400px;
            margin: 0 auto;
            padding: 2rem;
            display: grid;
            grid-template-columns: 400px 1fr 350px;
            gap: 2rem;
            min-height: calc(100vh - 100px);
        }

        .panel {
            background: rgba(255, 255, 255, 0.95);
            backdrop-filter: blur(10px);
            border-radius: 16px;
            padding: 1.5rem;
            box-shadow: 0 8px 32px rgba(0, 0, 0, 0.1);
            border: 1px solid rgba(255, 255, 255, 0.2);
        }
```

```
        .panel-header {
            display: flex;
            justify-content: space-between;
            align-items: center;
            margin-bottom: 1.5rem;
            padding-bottom: 1rem;
            border-bottom: 2px solid #f1f5f9;
        }

        .panel-title {
            font-size: 1.1rem;
            font-weight: 600;
            color: #334155;
            display: flex;
            align-items: center;
            gap: 0.5rem;
        }

        .form-group {
            margin-bottom: 1rem;
        }

        .form-label {
            display: block;
            margin-bottom: 0.5rem;
            font-weight: 500;
            color: #374151;
            font-size: 0.875rem;
        }

        .form-input, .form-select {
            width: 100%;
            padding: 0.75rem;
            border: 2px solid #e2e8f0;
            border-radius: 8px;
            font-size: 0.875rem;
            transition: all 0.2s ease;
            background: white;
        }

        .form-input:focus, .form-select:focus {
            outline: none;
            border-color: #667eea;
            box-shadow: 0 0 0 3px rgba(102, 126, 234, 0.1);
        }

        .form-row {
            display: grid;
            grid-template-columns: 1fr 1fr;
```

```
            gap: 1rem;
        }

        .btn {
            width: 100%;
            padding: 0.75rem 1.5rem;
            border: none;
            border-radius: 8px;
            font-weight: 600;
            font-size: 0.875rem;
            cursor: pointer;
            transition: all 0.2s ease;
            text-transform: uppercase;
            letter-spacing: 0.5px;
        }

        .btn-primary {
            background: linear-gradient(135deg, #667eea 0%, #764ba2
100%);
            color: white;
        }

        .btn-primary:hover {
            transform: translateY(-1px);
            box-shadow: 0 4px 12px rgba(102, 126, 234, 0.4);
        }

        .btn-danger {
            background: linear-gradient(135deg, #ef4444 0%, #dc2626
100%);
            color: white;
        }

        .btn-danger:hover {
            transform: translateY(-1px);
            box-shadow: 0 4px 12px rgba(239, 68, 68, 0.4);
        }

        .bbo-display {
            display: grid;
            grid-template-columns: 1fr 1fr;
            gap: 1rem;
            margin-bottom: 1.5rem;
        }

        .bbo-item {
            padding: 1rem;
            border-radius: 12px;
            text-align: center;
            position: relative;
```

```
            overflow: hidden;
        }

        .bbo-bid {
            background: linear-gradient(135deg, #10b981 0%, #059669
100%);
            color: white;
        }

        .bbo-ask {
            background: linear-gradient(135deg, #ef4444 0%, #dc2626
100%);
            color: white;
        }

        .bbo-label {
            font-size: 0.75rem;
            opacity: 0.9;
            margin-bottom: 0.25rem;
            text-transform: uppercase;
            letter-spacing: 0.5px;
        }

        .bbo-price {
            font-size: 1.25rem;
            font-weight: 700;
            font-family: 'Monaco', 'Menlo', monospace;
        }

        .orderbook {
            height: 400px;
            overflow-y: auto;
        }

        .orderbook-table {
            width: 100%;
            border-collapse: collapse;
        }

        .orderbook-header {
            background: #f8fafc;
            position: sticky;
            top: 0;
            z-index: 10;
        }

        .orderbook-header th {
            padding: 0.75rem 0.5rem;
            text-align: left;
            font-weight: 600;
```

```
            color: #475569;
            font-size: 0.75rem;
            text-transform: uppercase;
            letter-spacing: 0.5px;
            border-bottom: 2px solid #e2e8f0;
        }

        .orderbook-row {
            transition: background-color 0.2s ease;
        }

        .orderbook-row:hover {
            background-color: rgba(102, 126, 234, 0.05);
        }

        .orderbook-cell {
            padding: 0.5rem;
            font-family: 'Monaco', 'Menlo', monospace;
            font-size: 0.8rem;
            border-bottom: 1px solid #f1f5f9;
        }

        .asks .orderbook-cell:first-child {
            color: #dc2626;
            font-weight: 600;
        }

        .bids .orderbook-cell:first-child {
            color: #059669;
            font-weight: 600;
        }

        .trade-feed {
            height: 500px;
            overflow-y: auto;
        }

        .trade-item {
            padding: 1rem;
            border-bottom: 1px solid #f1f5f9;
            transition: all 0.3s ease;
            border-left: 4px solid transparent;
        }

        .trade-item:hover {
            background-color: #f8fafc;
            border-left-color: #667eea;
        }

        .trade-item.new-trade {
```

```
            animation: slideIn 0.5s ease, highlight 2s ease;
        }

        @keyframes slideIn {
            from {
                transform: translateX(-100%);
                opacity: 0;
            }
            to {
                transform: translateX(0);
                opacity: 1;
            }
        }

        @keyframes highlight {
            0%, 100% { background-color: transparent; }
            50% { background-color: rgba(102, 126, 234, 0.1); }
        }

        .trade-symbol {
            font-weight: 600;
            color: #334155;
        }

        .trade-price {
            font-family: 'Monaco', 'Menlo', monospace;
            font-weight: 700;
            margin-left: 0.5rem;
        }

        .trade-quantity {
            font-family: 'Monaco', 'Menlo', monospace;
            font-size: 0.9rem;
            margin-top: 0.25rem;
        }

        .trade-meta {
            font-size: 0.75rem;
            color: #64748b;
            margin-top: 0.5rem;
            display: flex;
            justify-content: space-between;
        }

        .aggressor-buy {
            color: #059669;
        }

        .aggressor-sell {
            color: #dc2626;
```

```
        }

        .stats-grid {
            display: grid;
            grid-template-columns: 1fr 1fr;
            gap: 1rem;
            margin-bottom: 1.5rem;
        }

        .stat-item {
            padding: 1rem;
            background: #f8fafc;
            border-radius: 8px;
            text-align: center;
        }

        .stat-value {
            font-size: 1.25rem;
            font-weight: 700;
            color: #334155;
            margin-bottom: 0.25rem;
        }

        .stat-label {
            font-size: 0.75rem;
            color: #64748b;
            text-transform: uppercase;
            letter-spacing: 0.5px;
        }

        .symbol-selector {
            margin-bottom: 1.5rem;
        }

        .alert {
            padding: 1rem;
            border-radius: 8px;
            margin-bottom: 1rem;
            font-size: 0.875rem;
        }

        .alert-success {
            background-color: #dcfce7;
            color: #166534;
            border: 1px solid #bbf7d0;
        }

        .alert-error {
            background-color: #fef2f2;
            color: #991b1b;
```

```
            border: 1px solid #fecaca;
        }

        .loading {
            opacity: 0.6;
            pointer-events: none;
        }

        .spinner {
            display: inline-block;
            width: 16px;
            height: 16px;
            border: 2px solid #f3f3f3;
            border-top: 2px solid #667eea;
            border-radius: 50%;
            animation: spin 1s linear infinite;
        }

        @keyframes spin {
            0% { transform: rotate(0deg); }
            100% { transform: rotate(360deg); }
        }

        /* Responsive Design */
        @media (max-width: 1200px) {
            .container {
                grid-template-columns: 1fr;
                gap: 1rem;
            }
        }

        @media (max-width: 768px) {
            .container {
                padding: 1rem;
            }

            .header-content {
                flex-direction: column;
                gap: 1rem;
            }

            .status-bar {
                gap: 1rem;
            }
        }
        .alert-info {
    background-color: #e0f2fe;
    color: #0c4a6e;
    border: 1px solid #7dd3fc;
}
```

```
    </style>
</head>
<body>
    <header class="header">
        <div class="header-content">
            <div class="logo">
                <i class="fas fa-chart-line"></i>
                Matching Engine Pro
            </div>
            <div class="status-bar">
                <div class="status-item">
                    <div class="status-label">Connection</div>
                    <div id="conn-status" class="status-value status-
disconnected">Disconnected</div>
                </div>
                <div class="status-item">
                    <div class="status-label">Latency</div>
                    <div id="latency" class="status-value">-- ms</div>
                </div>
                <div class="status-item">
                    <div class="status-label">Orders</div>
                    <div id="order-count" class="status-value">0</div>
                </div>
                <div class="status-item">
                    <div class="status-label">Trades</div>
                    <div id="trade-count" class="status-value">0</div>
                </div>
            </div>
        </div>
    </header>

    <div class="container">
        <!-- Order Entry Panel -->
        <div class="panel">
            <div class="panel-header">
                <div class="panel-title">
                    <i class="fas fa-plus-circle"></i>
                    Order Entry
                </div>
            </div>

            <div id="order-alert" class="alert" style="display:
none;"></div>

            <div class="symbol-selector">
                <label class="form-label">Trading Symbol</label>
                <select id="symbol" class="form-select">
                    <option value="BTC-USDT">BTC-USDT</option>
                    <option value="ETH-USDT">ETH-USDT</option>
                    <option value="BNB-USDT">BNB-USDT</option>
```

```
                    <option value="SOL-USDT">SOL-USDT</option>
                </select>
            </div>

            <div class="form-row">
                <div class="form-group">
                    <label class="form-label">Side</label>
                    <select id="side" class="form-select">
                        <option value="buy">Buy</option>
                        <option value="sell">Sell</option>
                    </select>
                </div>
                <div class="form-group">
                    <label class="form-label">Order Type</label>
                    <select id="order_type" class="form-select">
                        <option value="limit">Limit</option>
                        <option value="market">Market</option>
                        <option value="ioc">IOC</option>
                        <option value="fok">FOK</option>
                    </select>
                </div>
            </div>

            <div class="form-group">
                <label class="form-label">Quantity</label>
                <input id="quantity" type="number" step="any"
class="form-input" placeholder="e.g. 0.001">
            </div>

            <div class="form-group">
                <label class="form-label">Price (for Limit
orders)</label>
                <input id="price" type="number" step="any"
class="form-input" placeholder="e.g. 30000.00">
            </div>

            <div class="form-group">
                <label class="form-label">Client Order ID
(optional)</label>
                <input id="client_order_id" type="text" class="form-
input" placeholder="Custom order identifier">
            </div>

            <button id="submitBtn" class="btn btn-primary">
                <i class="fas fa-paper-plane"></i>
                Submit Order
            </button>

            <hr style="margin: 2rem 0; border: none; height: 1px;
background: #e2e8f0;">
```

```
            <div class="panel-header" style="margin-bottom: 1rem;">
                <div class="panel-title">
                    <i class="fas fa-times-circle"></i>
                    Cancel Order
                </div>
            </div>

            <div class="form-group">
                <label class="form-label">Order ID</label>
                <input id="cancel_order_id" type="text" class="form-
input" placeholder="Order ID to cancel">
            </div>

            <button id="cancelBtn" class="btn btn-danger">
                <i class="fas fa-ban"></i>
                Cancel Order
            </button>
        </div>

        <!-- Market Data Panel -->
        <div class="panel">
            <div class="panel-header">
                <div class="panel-title">
                    <i class="fas fa-chart-area"></i>
                    Market Data
                </div>
                <div id="current-symbol" style="font-weight: 600;
color: #667eea;">BTC-USDT</div>
            </div>

            <!-- Best Bid/Offer Display -->
            <div class="bbo-display">
                <div class="bbo-item bbo-bid">
                    <div class="bbo-label">Best Bid</div>
                    <div id="best_bid" class="bbo-price">--</div>
                </div>
                <div class="bbo-item bbo-ask">
                    <div class="bbo-label">Best Ask</div>
                    <div id="best_ask" class="bbo-price">--</div>
                </div>
            </div>

            <!-- Statistics Grid -->
            <div class="stats-grid">
                <div class="stat-item">
                    <div id="spread" class="stat-value">--</div>
                    <div class="stat-label">Spread</div>
                </div>
                <div class="stat-item">
```

```
                    <div id="mid_price" class="stat-value">--</div>
                    <div class="stat-label">Mid Price</div>
                </div>
            </div>

            <!-- Order Book -->
            <div class="panel-header" style="margin-bottom: 0.5rem;">
                <div class="panel-title" style="font-size: 1rem;">
                    <i class="fas fa-list"></i>
                    Order Book
                </div>
            </div>

            <div class="orderbook">
                <table class="orderbook-table">
                    <thead class="orderbook-header">
                        <tr>
                            <th>Price</th>
                            <th>Size</th>
                            <th>Total</th>
                        </tr>
                    </thead>
                    <tbody id="asks_table_body">
                        <!-- Asks will be populated here -->
                    </tbody>
                </table>

                <div style="text-align: center; padding: 0.5rem;
background: #f8fafc; font-weight: 600; color: #64748b; font-size:
0.8rem;">
                    --- SPREAD ---
                </div>

                <table class="orderbook-table">
                    <tbody id="bids_table_body">
                        <!-- Bids will be populated here -->
                    </tbody>
                </table>
            </div>
        </div>

        <!-- Trade Feed Panel -->
        <div class="panel">
            <div class="panel-header">
                <div class="panel-title">
                    <i class="fas fa-exchange-alt"></i>
                    Live Trades
                </div>
                <button id="clear-trades" style="padding: 0.25rem
0.5rem; border: none; background: #e2e8f0; border-radius: 4px; cursor:
```

```
pointer; font-size: 0.75rem;">
                    Clear
                </button>
            </div>

            <div class="trade-feed" id="trade_feed">
                <div style="text-align: center; color: #64748b;
padding: 2rem;">
                    <i class="fas fa-clock"></i>
                    <div style="margin-top: 0.5rem;">Waiting for
trades...</div>
                </div>
            </div>
        </div>
    </div>

    <!-- Toast Notifications Container -->
    <div id="toast-container" style="position: fixed; top: 20px;
right: 20px; z-index: 1000;"></div>

    <script
src="https://cdn.socket.io/4.5.4/socket.io.min.js"></script>
    <script src="./main.js"></script>
</body>
</html>
'''
```

frontend/main.js

```
'''
// frontend/main.js
// Enhanced cryptocurrency matching engine frontend with advanced
features

(() => {
    // Configuration
    const CONFIG = {
        API_HOST: "", // empty => same origin
        SOCKET_PATH: "/socket.io/",
        NAMESPACE: "/market",
        RECONNECT_ATTEMPTS: 5,
        RECONNECT_DELAY: 2000,
        PING_INTERVAL: 30000,
        MAX_TRADE_HISTORY: 100,
        MAX_ORDERBOOK_LEVELS: 20
    };

    // State management
    const state = {
        connected: false,
```

```
        currentSymbol: "BTC-USDT",
        orderCount: 0,
        tradeCount: 0,
        latency: 0,
        lastPingTime: 0,
        subscriptions: new Set(),
        orderBook: {
            bids: [],
            asks: [],
            lastUpdate: 0
        },
        trades: [],
        statistics: {
            spread: null,
            midPrice: null,
            volume24h: 0
        }
    };

    // DOM elements
    const elements = {
        // Connection status
        connStatus: document.getElementById("conn-status"),
        latency: document.getElementById("latency"),
        orderCount: document.getElementById("order-count"),
        tradeCount: document.getElementById("trade-count"),

        // Order form
        symbol: document.getElementById("symbol"),
        side: document.getElementById("side"),
        orderType: document.getElementById("order_type"),
        quantity: document.getElementById("quantity"),
        price: document.getElementById("price"),
        clientOrderId: document.getElementById("client_order_id"),
        submitBtn: document.getElementById("submitBtn"),
        orderAlert: document.getElementById("order-alert"),

        // Cancel form
        cancelOrderId: document.getElementById("cancel_order_id"),
        cancelBtn: document.getElementById("cancelBtn"),

        // Market data
        currentSymbol: document.getElementById("current-symbol"),
        bestBid: document.getElementById("best_bid"),
        bestAsk: document.getElementById("best_ask"),
        spread: document.getElementById("spread"),
        midPrice: document.getElementById("mid_price"),
        asksTableBody: document.getElementById("asks_table_body"),
        bidsTableBody: document.getElementById("bids_table_body"),
```

```
        // Trade feed
        tradeFeed: document.getElementById("trade_feed"),
        clearTrades: document.getElementById("clear-trades"),
        toastContainer: document.getElementById("toast-container")
    };

    // Socket.IO client
    let socket = null;
    let reconnectAttempts = 0;
    let pingInterval = null;

    // Initialize application
    function init() {
        setupEventListeners();
        connectSocket();
        startPingTimer();
        loadSymbols();

        // Set initial symbol
        updateSymbolDisplay();

        console.info("Matching Engine Frontend initialized");
    }

    // Socket connection management
    function connectSocket() {
        try {
            socket = io(CONFIG.API_HOST + CONFIG.NAMESPACE, {
                path: CONFIG.SOCKET_PATH,
                transports: ['websocket', 'polling'],
                timeout: 5000,
                forceNew: true
            });

            socket.on("connect", onSocketConnect);
            socket.on("disconnect", onSocketDisconnect);
            socket.on("connect_error", onSocketError);
            socket.on("reconnect", onSocketReconnect);

            // Market data events
            socket.on("connected", onWelcome);
            socket.on("l2_update", onL2Update);
            socket.on("trade", onTrade);
            socket.on("order_event", onOrderEvent);
            socket.on("subscribed", onSubscribed);
            socket.on("error", onSocketServerError);
            socket.on("pong", onPong);

        } catch (error) {
            console.error("Failed to initialize socket:", error);
```

```
            showToast("Connection failed", "error");
        }
    }

    // Socket event handlers
    function onSocketConnect() {
        console.info("Connected to matching engine:", socket.id);
        state.connected = true;
        reconnectAttempts = 0;
        updateConnectionStatus();

        // Subscribe to current symbol
        subscribeToSymbol(state.currentSymbol);
        subscribeToTrades();

        showToast("Connected to matching engine", "success");
    }

    function onSocketDisconnect(reason) {
        console.warn("Disconnected from server:", reason);
        state.connected = false;
        updateConnectionStatus();
        showToast("Connection lost", "error");

        // Attempt reconnection for client-side disconnects
        if (reason !== "io server disconnect" && reconnectAttempts <
CONFIG.RECONNECT_ATTEMPTS) {
            setTimeout(() => {
                reconnectAttempts++;
                console.info(`Reconnection attempt $
{reconnectAttempts}/${CONFIG.RECONNECT_ATTEMPTS}`);
                connectSocket();
            }, CONFIG.RECONNECT_DELAY * reconnectAttempts);
        }
    }

    function onSocketError(error) {
        console.error("Socket connection error:", error);
        showToast(`Connection error: ${error.message}`, "error");
    }

    function onSocketReconnect() {
        console.info("Reconnected to server");
        showToast("Reconnected successfully", "success");
    }

    function onWelcome(data) {
        console.info("Welcome message:", data);
        if (data.symbols && Array.isArray(data.symbols)) {
            updateSymbolOptions(data.symbols);
```

```
        }
    }

    function onL2Update(data) {
        if (!data || data.symbol !== state.currentSymbol) return;

        state.orderBook = {
            bids: data.bids || [],
            asks: data.asks || [],
            lastUpdate: data.timestamp || Date.now() / 1000
        };

        updateOrderBookDisplay();
        updateBBODisplay(data);
        updateStatistics();
    }

    function onTrade(trade) {
        if (!trade) return;

        state.tradeCount++;
        elements.tradeCount.textContent = state.tradeCount;

        // Add to trade history
        state.trades.unshift(trade);
        if (state.trades.length > CONFIG.MAX_TRADE_HISTORY) {
            state.trades = state.trades.slice(0,
CONFIG.MAX_TRADE_HISTORY);
        }

        // Update display
        addTradeToFeed(trade);

        // Play sound notification (optional)
        playTradeSound();
    }

    function onOrderEvent(orderData) {
        console.info("Order event:", orderData);
        // Could be used to update order status, show notifications,
etc.
    }

    function onSubscribed(data) {
        console.info("Subscription confirmed:", data);
        state.subscriptions.add(data.type + (data.symbol ? ":" +
data.symbol : ""));
    }

    function onSocketServerError(error) {
```

```
        console.error("Server error:", error);
        showToast(`Server error: ${error.message}`, "error");
    }

    function onPong(data) {
        if (state.lastPingTime > 0) {
            state.latency = Date.now() - state.lastPingTime;
            elements.latency.textContent = `${state.latency}ms`;
        }
    }

    // Subscription management
    function subscribeToSymbol(symbol) {
        if (socket && socket.connected) {
            socket.emit("subscribe", {
                type: "l2_updates",
                symbol: symbol
            });
        }
    }

    function subscribeToTrades() {
        if (socket && socket.connected) {
            socket.emit("subscribe", {
                type: "trades"
            });
        }
    }

    function unsubscribeFromSymbol(symbol) {
        if (socket && socket.connected) {
            socket.emit("unsubscribe", {
                type: "l2_updates",
                symbol: symbol
            });
        }
    }

    // UI Update functions
    function updateConnectionStatus() {
        if (state.connected) {
            elements.connStatus.textContent = "Connected";
            elements.connStatus.className = "status-value status-
connected";
        } else {
            elements.connStatus.textContent = "Disconnected";
            elements.connStatus.className = "status-value status-
disconnected";
            elements.latency.textContent = "-- ms";
        }
```

```
    }

    function updateSymbolDisplay() {
        elements.currentSymbol.textContent = state.currentSymbol;
    }

    function updateOrderBookDisplay() {
        const { bids, asks } = state.orderBook;

        // Clear existing data
        elements.asksTableBody.innerHTML = "";
        elements.bidsTableBody.innerHTML = "";

        // Add asks (reverse order to show best ask at bottom)
        const reversedAsks = [...asks].reverse().slice(0,
CONFIG.MAX_ORDERBOOK_LEVELS);
        reversedAsks.forEach(([price, size]) => {
            const total = calculateRunningTotal(reversedAsks, price,
true);
            addOrderBookRow(elements.asksTableBody, price, size,
total, "asks");
        });

        // Add bids (best bid at top)
        const topBids = bids.slice(0, CONFIG.MAX_ORDERBOOK_LEVELS);
        topBids.forEach(([price, size]) => {
            const total = calculateRunningTotal(topBids, price,
false);
            addOrderBookRow(elements.bidsTableBody, price, size,
total, "bids");
        });
    }

    function addOrderBookRow(tableBody, price, size, total, side) {
        const row = document.createElement("tr");
        row.className = `orderbook-row ${side}`;

        row.innerHTML = `
            <td class="orderbook-cell">${formatPrice(price)}</td>
            <td class="orderbook-cell">${formatQuantity(size)}</td>
            <td class="orderbook-cell">${formatQuantity(total)}</td>
        `;

        // Add click handler for quick order entry
        row.addEventListener("click", () => {
            elements.price.value = price;
            elements.side.value = side === "asks" ? "buy" : "sell";
        });

        tableBody.appendChild(row);
```

```
    }

    function calculateRunningTotal(levels, targetPrice, isAsk) {
        let total = 0;
        for (const [price, size] of levels) {
            if (isAsk ? parseFloat(price) <= parseFloat(targetPrice) :
parseFloat(price) >= parseFloat(targetPrice)) {
                total += parseFloat(size);
            }
            if (parseFloat(price) === parseFloat(targetPrice)) break;
        }
        return total;
    }

    function updateBBODisplay(data) {
        if (data.bids && data.bids.length > 0) {
            elements.bestBid.textContent = formatPrice(data.bids[0]
[0]);
        } else {
            elements.bestBid.textContent = "--";
        }

        if (data.asks && data.asks.length > 0) {
            elements.bestAsk.textContent = formatPrice(data.asks[0]
[0]);
        } else {
            elements.bestAsk.textContent = "--";
        }
    }

    function updateStatistics() {
        const { bids, asks } = state.orderBook;

        if (bids.length > 0 && asks.length > 0) {
            const bestBid = parseFloat(bids[0][0]);
            const bestAsk = parseFloat(asks[0][0]);

            state.statistics.spread = bestAsk - bestBid;
            state.statistics.midPrice = (bestBid + bestAsk) / 2;

            elements.spread.textContent =
formatPrice(state.statistics.spread);
            elements.midPrice.textContent =
formatPrice(state.statistics.midPrice);
        } else {
            elements.spread.textContent = "--";
            elements.midPrice.textContent = "--";
        }
    }
```

```
    function addTradeToFeed(trade) {
        const tradeElement = document.createElement("div");
        tradeElement.className = "trade-item new-trade";

        const aggressorClass = trade.aggressor_side === "buy" ?
"aggressor-buy" : "aggressor-sell";
        const tradeTime = new Date(trade.timestamp *
1000).toLocaleTimeString();

        tradeElement.innerHTML = `
            <div>
                <span class="trade-symbol">${trade.symbol}</span>
                <span class="trade-price ${aggressorClass}">$
{formatPrice(trade.price)}</span>
            </div>
            <div class="trade-quantity">
                <strong>${formatQuantity(trade.quantity)}</strong>
                <span style="margin-left: 0.5rem; color: #64748b;">
                    ${trade.aggressor_side.toUpperCase()}
                </span>
            </div>
            <div class="trade-meta">
                <span>ID: ${trade.trade_id.substring(0, 12)}...</span>
                <span>${tradeTime}</span>
            </div>
        `;

        // Add to beginning of feed
        if (elements.tradeFeed.children.length === 0 ||

elements.tradeFeed.children[0].textContent.includes("Waiting for
trades")) {
            elements.tradeFeed.innerHTML = "";
        }

        elements.tradeFeed.insertBefore(tradeElement,
elements.tradeFeed.firstChild);

        // Limit trade history in DOM
        const tradeItems = elements.tradeFeed.children;
        if (tradeItems.length > CONFIG.MAX_TRADE_HISTORY) {
            for (let i = CONFIG.MAX_TRADE_HISTORY; i <
tradeItems.length; i++) {
                tradeItems[i].remove();
            }
        }
    }

    // Order management
    async function submitOrder() {
```

```
        if (!state.connected) {
            showAlert("Not connected to server", "error");
            return;
        }

        const orderData = {
            symbol: elements.symbol.value || state.currentSymbol,
            side: elements.side.value,
            order_type: elements.orderType.value,
            quantity: elements.quantity.value
        };

        // Add price for limit orders
        if (elements.orderType.value === "limit") {
            if (!elements.price.value) {
                showAlert("Price is required for limit orders",
"error");
                return;
            }
            orderData.price = elements.price.value;
        }

        // Add client order ID if provided
        if (elements.clientOrderId.value) {
            orderData.client_order_id = elements.clientOrderId.value;
        }

        try {
            setLoading(elements.submitBtn, true);
            showAlert("Submitting order...", "info");

            const response = await fetch(`${CONFIG.API_HOST}/order`, {
                method: "POST",
                headers: {
                    "Content-Type": "application/json"
                },
                body: JSON.stringify(orderData)
            });

            const result = await response.json();

            if (response.ok) {
                showAlert(`Order submitted: ${result.order_id}`,
"success");
                state.orderCount++;
                elements.orderCount.textContent = state.orderCount;

                // Clear form
                elements.quantity.value = "";
                elements.clientOrderId.value = "";
```

```
                // Show trades if any
                if (result.trades && result.trades.length > 0) {
                    result.trades.forEach(trade => onTrade(trade));
                }

                showToast(`Order ${result.status}: $
{result.order_id.substring(0, 8)}...`, "success");
            } else {
                showAlert(`Order failed: ${result.error}`, "error");
                showToast("Order submission failed", "error");
            }

        } catch (error) {
            console.error("Order submission error:", error);
            showAlert(`Network error: ${error.message}`, "error");
            showToast("Network error occurred", "error");
        } finally {
            setLoading(elements.submitBtn, false);
        }
    }

    async function cancelOrder() {
        const orderId = elements.cancelOrderId.value.trim();
        if (!orderId) {
            showAlert("Please enter an Order ID", "error");
            return;
        }

        try {
            setLoading(elements.cancelBtn, true);

            const response = await fetch(`${CONFIG.API_HOST}/cancel`,
{
                method: "POST",
                headers: {
                    "Content-Type": "application/json"
                },
                body: JSON.stringify({
                    symbol: state.currentSymbol,
                    order_id: orderId
                })
            });

            const result = await response.json();

            if (response.ok && result.success) {
                showAlert(`Order cancelled: ${orderId}`, "success");
                elements.cancelOrderId.value = "";
                showToast("Order cancelled successfully", "success");
```

```
            } else {
                showAlert(`Cancel failed: ${result.error || "Unknown
error"}`, "error");
                showToast("Order cancellation failed", "error");
            }

        } catch (error) {
            console.error("Cancel error:", error);
            showAlert(`Network error: ${error.message}`, "error");
        } finally {
            setLoading(elements.cancelBtn, false);
        }
    }

    // Event listeners setup
    function setupEventListeners() {
        // Order submission
        elements.submitBtn.addEventListener("click", submitOrder);
        elements.cancelBtn.addEventListener("click", cancelOrder);

        // Symbol change
        elements.symbol.addEventListener("change", (e) => {
            const oldSymbol = state.currentSymbol;
            state.currentSymbol = e.target.value;

            // Update subscriptions
            if (state.connected) {
                unsubscribeFromSymbol(oldSymbol);
                subscribeToSymbol(state.currentSymbol);
            }

            updateSymbolDisplay();
            clearOrderBook();
            clearTrades();
        });

        // Order type change
        elements.orderType.addEventListener("change", (e) => {
            const isLimit = e.target.value === "limit";
            elements.price.disabled = !isLimit;
            if (!isLimit) {
                elements.price.value = "";
            }
        });

        // Clear trades button
        elements.clearTrades.addEventListener("click", clearTrades);

        // Keyboard shortcuts
        document.addEventListener("keydown", (e) => {
```

```
            if (e.ctrlKey || e.metaKey) {
                switch (e.key) {
                    case "Enter":
                        e.preventDefault();
                        submitOrder();
                        break;
                    case "Escape":
                        clearForm();
                        break;
                }
            }
        });

        // Form validation
        elements.quantity.addEventListener("input", validateQuantity);
        elements.price.addEventListener("input", validatePrice);
    }

    // Utility functions
    function formatPrice(price) {
        const num = parseFloat(price);
        return num.toLocaleString(undefined, {
            minimumFractionDigits: 2,
            maximumFractionDigits: 8
        });
    }

    function formatQuantity(quantity) {
        const num = parseFloat(quantity);
        return num.toLocaleString(undefined, {
            minimumFractionDigits: 0,
            maximumFractionDigits: 8
        });
    }

    function showAlert(message, type) {
        elements.orderAlert.textContent = message;
        elements.orderAlert.className = `alert alert-${type}`;
        elements.orderAlert.style.display = "block";

        // Auto-hide success messages
        if (type === "success") {
            setTimeout(() => {
                elements.orderAlert.style.display = "none";
            }, 3000);
        }
    }

    function showToast(message, type = "info") {
        const toast = document.createElement("div");
```

```
        toast.className = `alert alert-${type}`;
        toast.style.cssText = `
            position: relative;
            margin-bottom: 0.5rem;
            animation: slideInRight 0.3s ease;
            max-width: 300px;
        `;
        toast.textContent = message;

        elements.toastContainer.appendChild(toast);

        // Auto-remove toast
        setTimeout(() => {
            toast.style.animation = "slideOutRight 0.3s ease";
            setTimeout(() => toast.remove(), 300);
        }, 3000);
    }

    function setLoading(button, loading) {
        if (loading) {
            button.disabled = true;
            button.classList.add("loading");
            const spinner = document.createElement("span");
            spinner.className = "spinner";
            button.prepend(spinner);
        } else {
            button.disabled = false;
            button.classList.remove("loading");
            const spinner = button.querySelector(".spinner");
            if (spinner) spinner.remove();
        }
    }

    function clearOrderBook() {
        elements.asksTableBody.innerHTML = "";
        elements.bidsTableBody.innerHTML = "";
        elements.bestBid.textContent = "--";
        elements.bestAsk.textContent = "--";
        elements.spread.textContent = "--";
        elements.midPrice.textContent = "--";
    }

    function clearTrades() {
        elements.tradeFeed.innerHTML = `
            <div style="text-align: center; color: #64748b; padding:
2rem;">
                <i class="fas fa-clock"></i>
                <div style="margin-top: 0.5rem;">Waiting for
trades...</div>
            </div>
```

```
        `;
        state.trades = [];
    }

    function clearForm() {
        elements.quantity.value = "";
        elements.price.value = "";
        elements.clientOrderId.value = "";
        elements.cancelOrderId.value = "";
        elements.orderAlert.style.display = "none";
    }

    function validateQuantity(e) {
        const value = parseFloat(e.target.value);
        if (isNaN(value) || value <= 0) {
            e.target.setCustomValidity("Quantity must be a positive
number");
        } else {
            e.target.setCustomValidity("");
        }
    }

    function validatePrice(e) {
        if (elements.orderType.value === "limit") {
            const value = parseFloat(e.target.value);
            if (isNaN(value) || value <= 0) {
                e.target.setCustomValidity("Price must be a positive
number");
            } else {
                e.target.setCustomValidity("");
            }
        }
    }

    function startPingTimer() {
        pingInterval = setInterval(() => {
            if (socket && socket.connected) {
                state.lastPingTime = Date.now();
                socket.emit("ping");
            }
        }, CONFIG.PING_INTERVAL);
    }

    function loadSymbols() {
        // Load available symbols from API
        fetch(`${CONFIG.API_HOST}/symbols`)
            .then(response => response.json())
            .then(data => {
                if (data.symbols) {
                    updateSymbolOptions(data.symbols);
```

```
                }
            })
            .catch(error => {
                console.warn("Failed to load symbols:", error);
            });
    }

    function updateSymbolOptions(symbols) {
        elements.symbol.innerHTML = "";
        symbols.forEach(symbol => {
            const option = document.createElement("option");
            option.value = symbol;
            option.textContent = symbol;
            option.selected = symbol === state.currentSymbol;
            elements.symbol.appendChild(option);
        });
    }

    function playTradeSound() {
        // Optional: play sound notification for trades
        try {
            const audio = new
Audio("data:audio/wav;base64,UklGRnoGAABXQVZFZm10IBAAAAABAAEAQB8AAEAfA
AABAAgAZGF0YQoGAACBhYqFbF1fdJivrJBhNjVgodDbq2EcBj+a2/
LDciUFLIHO8tiJNwgZaLvt559NEAxQp+PwtmMcBjiR1/
LMeSwFJHfH8N2QQAoUXrTp66hVFApGn+HyvmAZBjiB0vLLdSEGJ3PN8thzEgEfa7bv3JFB
Cw0=" );
            audio.volume = 0.1;
            audio.play().catch(() => {}); // Ignore errors
        } catch (e) {
            // Ignore audio errors
        }
    }

    // Initialize on DOM ready
    if (document.readyState === "loading") {
        document.addEventListener("DOMContentLoaded", init);
    } else {
        init();
    }

    // Cleanup on page unload
    window.addEventListener("beforeunload", () => {
        if (socket && socket.connected) {
            socket.disconnect();
        }
        if (pingInterval) {
            clearInterval(pingInterval);
        }
    });
```

```
    // Add CSS animations dynamically
    const style = document.createElement("style");
    style.textContent = `
        @keyframes slideInRight {
            from {
                transform: translateX(100%);
                opacity: 0;
            }
            to {
                transform: translateX(0);
                opacity: 1;
            }
        }

        @keyframes slideOutRight {
            from {
                transform: translateX(0);
                opacity: 1;
            }
            to {
                transform: translateX(100%);
                opacity: 0;
            }
        }

        .orderbook-row:hover {
            cursor: pointer;
            background-color: rgba(102, 126, 234, 0.05) !important;
        }

        .form-input:invalid {
            border-color: #ef4444;
        }

        .form-input:valid {
            border-color: #10b981;
        }
    `;
    document.head.appendChild(style);

    // Export functions for debugging (development only)
    if (window.location.hostname === "localhost" ||
window.location.hostname === "127.0.0.1") {
        window.matchingEngineDebug = {
            state,
            socket,
            submitOrder,
            cancelOrder,
```

```
            connectSocket,
            showToast,
            clearTrades
        };
        console.info("Debug functions available at
window.matchingEngineDebug");
    }

})();
'''
```

tests/advanced_load_test.py

```python
# advanced_load_test.py
"""
Advanced Load Testing Suite for Cryptocurrency Matching Engine
Supports high concurrency with detailed metrics and monitoring
"""

import asyncio
import aiohttp
import time
import json
import random
import statistics
import threading
from dataclasses import dataclass, field
from typing import List, Dict, Any, Optional
from collections import defaultdict, deque
import argparse
import logging
from datetime import datetime, timedelta
import csv
import signal
import sys

# Configure logging
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s [%(levelname)s] %(message)s'
)
logger = logging.getLogger(__name__)

@dataclass
class TestResult:
    """Individual test result"""
    timestamp: float
    duration_ms: float
    success: bool
    status_code: Optional[int] = None
```

```
    error: Optional[str] = None
    response_size: int = 0
    order_id: Optional[str] = None
    trades: int = 0

@dataclass
class MetricsCollector:
    """Comprehensive metrics collection"""
    start_time: float = field(default_factory=time.time)
    end_time: Optional[float] = None

    # Request metrics
    total_requests: int = 0
    successful_requests: int = 0
    failed_requests: int = 0

    # Latency metrics
    response_times: List[float] = field(default_factory=list)
    min_latency: float = float('inf')
    max_latency: float = 0

    # Throughput metrics
    requests_per_second: List[float] = field(default_factory=list)

    # Business metrics
    orders_submitted: int = 0
    trades_executed: int = 0
    total_volume: float = 0

    # Error tracking
    error_types: Dict[str, int] = field(default_factory=lambda:
defaultdict(int))
    status_codes: Dict[int, int] = field(default_factory=lambda:
defaultdict(int))

    # Concurrency metrics
    active_connections: int = 0
    peak_connections: int = 0

    def add_result(self, result: TestResult):
        """Add a test result to metrics"""
        self.total_requests += 1

        if result.success:
            self.successful_requests += 1
            self.response_times.append(result.duration_ms)
            self.min_latency = min(self.min_latency,
result.duration_ms)
            self.max_latency = max(self.max_latency,
```

```
result.duration_ms)

            if result.order_id:
                self.orders_submitted += 1

            self.trades_executed += result.trades
        else:
            self.failed_requests += 1
            if result.error:
                self.error_types[result.error] += 1

        if result.status_code:
            self.status_codes[result.status_code] += 1

    def get_percentiles(self, percentiles: List[float] = None) ->
Dict[str, float]:
        """Calculate response time percentiles"""
        if not self.response_times:
            return {}

        if percentiles is None:
            percentiles = [50, 75, 90, 95, 99, 99.9]

        sorted_times = sorted(self.response_times)
        result = {}

        for p in percentiles:
            index = int(len(sorted_times) * p / 100) - 1
            index = max(0, min(index, len(sorted_times) - 1))
            result[f"p{p}"] = sorted_times[index]

        return result

    def get_summary(self) -> Dict[str, Any]:
        """Get comprehensive test summary"""
        duration = (self.end_time or time.time()) - self.start_time

        percentiles = self.get_percentiles()

        return {
            "test_duration_seconds": round(duration, 2),
            "total_requests": self.total_requests,
            "successful_requests": self.successful_requests,
            "failed_requests": self.failed_requests,
            "success_rate": round((self.successful_requests /
max(self.total_requests, 1)) * 100, 2),
            "requests_per_second": round(self.total_requests /
max(duration, 1), 2),
            "orders_submitted": self.orders_submitted,
```

```
            "trades_executed": self.trades_executed,
            "average_latency_ms":
round(statistics.mean(self.response_times) if self.response_times else
0, 2),
            "min_latency_ms": round(self.min_latency if
self.min_latency != float('inf') else 0, 2),
            "max_latency_ms": round(self.max_latency, 2),
            "median_latency_ms":
round(statistics.median(self.response_times) if self.response_times
else 0, 2),
            "percentiles": {k: round(v, 2) for k, v in
percentiles.items()},
            "peak_connections": self.peak_connections,
            "error_types": dict(self.error_types),
            "status_codes": dict(self.status_codes)
        }


class LoadTester:
    """High-performance async load tester"""

    def __init__(self, base_url: str = "http://localhost:5000",
                 max_connections: int = 1000, timeout: int = 30):
        self.base_url = base_url
        self.max_connections = max_connections
        self.timeout = timeout
        self.session: Optional[aiohttp.ClientSession] = None
        self.metrics = MetricsCollector()
        self.semaphore: Optional[asyncio.Semaphore] = None
        self.running = True

        # Test data generators
        self.symbols = ["BTC-USDT", "ETH-USDT", "BNB-USDT", "SOL-
USDT"]
        self.order_types = ["limit", "market", "ioc", "fok"]
        self.sides = ["buy", "sell"]

    async def __aenter__(self):
        """Async context manager entry"""
        connector = aiohttp.TCPConnector(
            limit=self.max_connections,
            limit_per_host=self.max_connections,
            ttl_dns_cache=300,
            use_dns_cache=True,
        )

        timeout_config = aiohttp.ClientTimeout(total=self.timeout)

        self.session = aiohttp.ClientSession(
```

```
            connector=connector,
            timeout=timeout_config,
            headers={"Content-Type": "application/json"}
        )

        self.semaphore = asyncio.Semaphore(self.max_connections)

        return self

    async def __aexit__(self, exc_type, exc_val, exc_tb):
        """Async context manager exit"""
        if self.session:
            await self.session.close()

    def generate_order_data(self) -> Dict[str, Any]:
        """Generate realistic order data"""
        symbol = random.choice(self.symbols)
        side = random.choice(self.sides)
        order_type = random.choice(self.order_types)

        # Generate realistic quantities
        quantity = round(random.uniform(0.001, 1.0), 6)

        order_data = {
            "symbol": symbol,
            "side": side,
            "order_type": order_type,
            "quantity": str(quantity)
        }

        # Add price for limit orders
        if order_type in ["limit", "ioc", "fok"]:
            base_prices = {
                "BTC-USDT": 30000,
                "ETH-USDT": 2000,
                "BNB-USDT": 300,
                "SOL-USDT": 100
            }

            base_price = base_prices.get(symbol, 30000)
            # Add some price variation
            price_variation = random.uniform(0.95, 1.05)
            price = round(base_price * price_variation, 2)
            order_data["price"] = str(price)

        return order_data

    async def submit_single_order(self) -> TestResult:
        """Submit a single order and measure performance"""
```

```
        start_time = time.time()

        async with self.semaphore:
            self.metrics.active_connections += 1
            self.metrics.peak_connections = max(
                self.metrics.peak_connections,
                self.metrics.active_connections
            )

            try:
                order_data = self.generate_order_data()

                async with self.session.post(
                    f"{self.base_url}/order",
                    json=order_data
                ) as response:
                    response_text = await response.text()
                    duration_ms = (time.time() - start_time) * 1000

                    result = TestResult(
                        timestamp=start_time,
                        duration_ms=duration_ms,
                        success=response.status == 200,
                        status_code=response.status,
                        response_size=len(response_text)
                    )

                    if response.status == 200:
                        try:
                            response_data = json.loads(response_text)
                            result.order_id =
response_data.get("order_id")
                            result.trades =
len(response_data.get("trades", []))
                        except json.JSONDecodeError:
                            result.error = "Invalid JSON response"
                            result.success = False
                    else:
                        result.error = f"HTTP {response.status}"

                    return result

            except asyncio.TimeoutError:
                duration_ms = (time.time() - start_time) * 1000
                return TestResult(
                    timestamp=start_time,
                    duration_ms=duration_ms,
                    success=False,
                    error="Timeout"
```

```
                )
            except Exception as e:
                duration_ms = (time.time() - start_time) * 1000
                return TestResult(
                    timestamp=start_time,
                    duration_ms=duration_ms,
                    success=False,
                    error=str(e)
                )
            finally:
                self.metrics.active_connections -= 1

    async def run_constant_rate_test(self, rate_per_second: int,
duration_seconds: int):
        """Run test at constant rate for specified duration"""
        logger.info(f"Starting constant rate test: {rate_per_second}
req/sec for {duration_seconds}s")

        end_time = time.time() + duration_seconds
        interval = 1.0 / rate_per_second

        tasks = []

        while time.time() < end_time and self.running:
            # Start new request
            task = asyncio.create_task(self.submit_single_order())
            tasks.append(task)

            # Process completed tasks
            done_tasks = [t for t in tasks if t.done()]
            for task in done_tasks:
                try:
                    result = await task
                    self.metrics.add_result(result)
                except Exception as e:
                    logger.error(f"Task error: {e}")
                tasks.remove(task)

            # Rate limiting
            await asyncio.sleep(interval)

        # Wait for remaining tasks
        if tasks:
            logger.info(f"Waiting for {len(tasks)} remaining
requests...")
            results = await asyncio.gather(*tasks,
return_exceptions=True)

            for result in results:
```

```
                if isinstance(result, TestResult):
                    self.metrics.add_result(result)
                elif isinstance(result, Exception):
                    logger.error(f"Final task error: {result}")

        self.metrics.end_time = time.time()
        logger.info("Constant rate test completed")

    async def run_spike_test(self, peak_rate: int, spike_duration:
int,
                             baseline_rate: int, total_duration: int):
        """Run spike test with sudden traffic increase"""
        logger.info(f"Starting spike test: baseline
{baseline_rate}/sec, spike to {peak_rate}/sec")

        start_time = time.time()
        spike_start = start_time + (total_duration - spike_duration) /
2
        spike_end = spike_start + spike_duration

        while time.time() - start_time < total_duration and
self.running:
            current_time = time.time()

            # Determine current rate
            if spike_start <= current_time <= spike_end:
                current_rate = peak_rate
            else:
                current_rate = baseline_rate

            # Submit request
            task = asyncio.create_task(self.submit_single_order())
            result = await task
            self.metrics.add_result(result)

            # Rate limiting
            interval = 1.0 / current_rate
            await asyncio.sleep(interval)

        self.metrics.end_time = time.time()
        logger.info("Spike test completed")

    def stop(self):
        """Stop the test gracefully"""
        self.running = False
        logger.info("Stopping load test...")


class MetricsReporter:
```

```
    """Advanced metrics reporting and analysis"""

    @staticmethod
    def print_summary(metrics: MetricsCollector):
        """Print comprehensive test summary"""
        summary = metrics.get_summary()

        print("\n" + "="*80)
        print("LOAD TEST RESULTS")
        print("="*80)

        # Test Overview
        print(f"Test Duration: {summary['test_duration_seconds']}s")
        print(f"Total Requests: {summary['total_requests']:,}")
        print(f"Successful: {summary['successful_requests']:,}
({summary['success_rate']}%)")
        print(f"Failed: {summary['failed_requests']:,}")
        print(f"Requests/Second:
{summary['requests_per_second']:,.2f}")

        # Business Metrics
        print(f"\nBUSINESS METRICS:")
        print(f"Orders Submitted: {summary['orders_submitted']:,}")
        print(f"Trades Executed: {summary['trades_executed']:,}")
        print(f"Trade Rate: {(summary['trades_executed'] /
max(summary['orders_submitted'], 1) * 100):.2f}%")

        # Latency Metrics
        print(f"\nLATENCY METRICS (ms):")
        print(f"Average: {summary['average_latency_ms']}")
        print(f"Median: {summary['median_latency_ms']}")
        print(f"Min: {summary['min_latency_ms']}")
        print(f"Max: {summary['max_latency_ms']}")

        print("\nPERCENTILES (ms):")
        for percentile, value in summary['percentiles'].items():
            print(f"  {percentile}: {value}")

        # Connection Metrics
        print(f"\nCONCURRENCY:")
        print(f"Peak Connections: {summary['peak_connections']}")

        # Error Analysis
        if summary['error_types']:
            print(f"\nERROR BREAKDOWN:")
            for error, count in summary['error_types'].items():
                print(f"  {error}: {count}")

        if summary['status_codes']:
```

```
            print(f"\nSTATUS CODES:")
            for code, count in summary['status_codes'].items():
                print(f"  {code}: {count}")

        print("="*80)

    @staticmethod
    def save_detailed_report(metrics: MetricsCollector, filename: str
= None):
        """Save detailed CSV report"""
        if filename is None:
            timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
            filename = f"load_test_report_{timestamp}.csv"

        summary = metrics.get_summary()

        with open(filename, 'w', newline='') as csvfile:
            writer = csv.writer(csvfile)

            # Write summary
            writer.writerow(["SUMMARY"])
            for key, value in summary.items():
                if isinstance(value, dict):
                    writer.writerow([key, ""])
                    for sub_key, sub_value in value.items():
                        writer.writerow([f"  {sub_key}", sub_value])
                else:
                    writer.writerow([key, value])

        logger.info(f"Detailed report saved to {filename}")


async def main():
    """Main test runner"""
    parser = argparse.ArgumentParser(description="Advanced Load
Testing for Matching Engine")
    parser.add_argument("--url", default="http://localhost:5000",
help="API base URL")
    parser.add_argument("--rate", type=int, default=100,
help="Requests per second")
    parser.add_argument("--duration", type=int, default=60, help="Test
duration in seconds")
    parser.add_argument("--connections", type=int, default=100,
help="Max concurrent connections")
    parser.add_argument("--test-type", choices=["constant", "spike"],
default="constant", help="Test type")
    parser.add_argument("--spike-rate", type=int, default=200,
help="Peak rate for spike test")
    parser.add_argument("--spike-duration", type=int, default=30,
```

```
help="Spike duration in seconds")
    parser.add_argument("--baseline-rate", type=int, default=100,
help="Baseline rate for spike test")
    parser.add_argument("--report", help="Save detailed report to
file")

    args = parser.parse_args()

    # Setup signal handler for graceful shutdown
    def signal_handler(signum, frame):
        logger.info("Received interrupt signal, shutting down...")
        sys.exit(0)

    signal.signal(signal.SIGINT, signal_handler)

    async with LoadTester(args.url, args.connections) as tester:
        try:
            if args.test_type == "constant":
                await tester.run_constant_rate_test(args.rate,
args.duration)
            elif args.test_type == "spike":
                await tester.run_spike_test(
                    args.spike_rate,
                    args.spike_duration,
                    args.baseline_rate,
                    args.duration
                )

            # Print results
            MetricsReporter.print_summary(tester.metrics)

            # Save detailed report if requested
            if args.report:
                MetricsReporter.save_detailed_report(tester.metrics,
args.report)

        except KeyboardInterrupt:
            logger.info("Test interrupted by user")
            tester.stop()
        except Exception as e:
            logger.error(f"Test failed: {e}")
            raise


if __name__ == "__main__":
    asyncio.run(main())
```

tests/monitoring_suite.py

```
# monitoring_suite.py
"""
Real-time monitoring and profiling tools for the matching engine
"""

import psutil
import time
import threading
import json
import requests
import matplotlib.pyplot as plt
from collections import deque, defaultdict
from datetime import datetime
import numpy as np

class SystemMonitor:
    """Monitor system resources during load testing"""

    def __init__(self, api_url: str = "http://localhost:5000",
interval: float = 1.0):
        self.api_url = api_url
        self.interval = interval
        self.running = False

        # Metrics storage
        self.timestamps = deque(maxlen=1000)
        self.cpu_usage = deque(maxlen=1000)
        self.memory_usage = deque(maxlen=1000)
        self.network_io = deque(maxlen=1000)
        self.disk_io = deque(maxlen=1000)

        # API metrics
        self.response_times = deque(maxlen=1000)
        self.active_connections = deque(maxlen=1000)
        self.orders_per_sec = deque(maxlen=1000)
        self.trades_per_sec = deque(maxlen=1000)

        # Process monitoring
        self.process = None
        self.find_matching_engine_process()

    def find_matching_engine_process(self):
        """Find the matching engine process"""
        for proc in psutil.process_iter(['pid', 'name', 'cmdline']):
            try:
                if 'python' in proc.info['name'].lower():
                    cmdline = ' '.join(proc.info['cmdline'])
                    if 'app.py' in cmdline or 'matching' in cmdline:
```

```
                        self.process =
psutil.Process(proc.info['pid'])
                        print(f"Found matching engine process: PID
{proc.info['pid']}")
                        break
            except (psutil.NoSuchProcess, psutil.AccessDenied):
                continue

    def start_monitoring(self):
        """Start monitoring in background thread"""
        self.running = True
        self.monitor_thread =
threading.Thread(target=self._monitor_loop)
        self.monitor_thread.daemon = True
        self.monitor_thread.start()
        print("System monitoring started")

    def stop_monitoring(self):
        """Stop monitoring"""
        self.running = False
        if hasattr(self, 'monitor_thread'):
            self.monitor_thread.join()
        print("System monitoring stopped")

    def _monitor_loop(self):
        """Main monitoring loop"""
        prev_net_io = psutil.net_io_counters()
        prev_disk_io = psutil.disk_io_counters()

        while self.running:
            timestamp = time.time()

            # System metrics
            cpu_percent = psutil.cpu_percent(interval=0.1)
            memory = psutil.virtual_memory()

            # Network I/O
            net_io = psutil.net_io_counters()
            net_speed = ((net_io.bytes_sent + net_io.bytes_recv) -
                         (prev_net_io.bytes_sent +
prev_net_io.bytes_recv)) / self.interval / 1024 / 1024  # MB/s
            prev_net_io = net_io

            # Disk I/O
            disk_io = psutil.disk_io_counters()
            if disk_io and prev_disk_io:
                disk_speed = ((disk_io.read_bytes +
disk_io.write_bytes) -
                              (prev_disk_io.read_bytes +
prev_disk_io.write_bytes)) / self.interval / 1024 / 1024  # MB/s
```

```
                prev_disk_io = disk_io
            else:
                disk_speed = 0

            # API health check
            api_response_time = self._check_api_health()

            # Store metrics
            self.timestamps.append(timestamp)
            self.cpu_usage.append(cpu_percent)
            self.memory_usage.append(memory.percent)
            self.network_io.append(net_speed)
            self.disk_io.append(disk_speed)
            self.response_times.append(api_response_time)

            time.sleep(self.interval)

    def _check_api_health(self) -> float:
        """Check API response time"""
        try:
            start_time = time.time()
            response = requests.get(f"{self.api_url}/health",
timeout=5)
            response_time = (time.time() - start_time) * 1000  # ms

            if response.status_code == 200:
                data = response.json()
                # Extract metrics if available
                return response_time
            return response_time
        except Exception:
            return -1  # Indicate failure

    def get_current_stats(self) -> dict:
        """Get current system statistics"""
        if not self.timestamps:
            return {}

        return {
            "timestamp": self.timestamps[-1],
            "cpu_percent": self.cpu_usage[-1],
            "memory_percent": self.memory_usage[-1],
            "network_io_mbps": self.network_io[-1],
            "disk_io_mbps": self.disk_io[-1],
            "api_response_time_ms": self.response_times[-1],
            "process_info": self._get_process_info()
        }

    def _get_process_info(self) -> dict:
        """Get process-specific information"""
```

```
        if not self.process:
            return {}

        try:
            return {
                "cpu_percent": self.process.cpu_percent(),
                "memory_mb": self.process.memory_info().rss / 1024 /
1024,
                "num_threads": self.process.num_threads(),
                "num_fds": self.process.num_fds() if
hasattr(self.process, 'num_fds') else 0,
                "connections": len(self.process.connections()) if
hasattr(self.process, 'connections') else 0
            }
        except (psutil.NoSuchProcess, psutil.AccessDenied):
            return {}

    def plot_metrics(self, save_path: str = None):
        """Plot system metrics"""
        if len(self.timestamps) < 2:
            print("Not enough data to plot")
            return

        fig, axes = plt.subplots(2, 3, figsize=(15, 10))
        fig.suptitle('System Performance Metrics')

        times = [(t - self.timestamps[0]) for t in self.timestamps]

        # CPU Usage
        axes[0, 0].plot(times, list(self.cpu_usage))
        axes[0, 0].set_title('CPU Usage (%)')
        axes[0, 0].set_ylabel('Percent')

        # Memory Usage
        axes[0, 1].plot(times, list(self.memory_usage))
        axes[0, 1].set_title('Memory Usage (%)')
        axes[0, 1].set_ylabel('Percent')

        # Network I/O
        axes[0, 2].plot(times, list(self.network_io))
        axes[0, 2].set_title('Network I/O (MB/s)')
        axes[0, 2].set_ylabel('MB/s')

        # Disk I/O
        axes[1, 0].plot(times, list(self.disk_io))
        axes[1, 0].set_title('Disk I/O (MB/s)')
        axes[1, 0].set_ylabel('MB/s')

        # API Response Time
        valid_response_times = [rt for rt in self.response_times if rt
```

```
> 0]
        if valid_response_times:
            axes[1, 1].plot(times[-len(valid_response_times):],
valid_response_times)
            axes[1, 1].set_title('API Response Time (ms)')
            axes[1, 1].set_ylabel('Milliseconds')

        # Resource Utilization Summary
        if self.process:
            proc_info = self._get_process_info()
            axes[1, 2].bar(['CPU %', 'Memory MB', 'Threads',
'Connections'],
                           [proc_info.get('cpu_percent', 0),
                            proc_info.get('memory_mb', 0),
                            proc_info.get('num_threads', 0),
                            proc_info.get('connections', 0)])
            axes[1, 2].set_title('Process Statistics')

        for ax in axes.flat:
            ax.set_xlabel('Time (seconds)')
            ax.grid(True, alpha=0.3)

        plt.tight_layout()

        if save_path:
            plt.savefig(save_path, dpi=300, bbox_inches='tight')
            print(f"Metrics plot saved to {save_path}")

        plt.show()


class LoadTestOrchestrator:
    """Orchestrate multiple load tests with monitoring"""

    def __init__(self, api_url: str = "http://localhost:5000"):
        self.api_url = api_url
        self.monitor = SystemMonitor(api_url)
        self.test_results = []

    async def run_comprehensive_test_suite(self):
        """Run a comprehensive suite of load tests"""
        print("Starting comprehensive load test suite...")

        # Start monitoring
        self.monitor.start_monitoring()

        test_scenarios = [
            ("Warmup", {"rate": 10, "duration": 30}),
            ("Low Load", {"rate": 100, "duration": 60}),
            ("Medium Load", {"rate": 500, "duration": 120}),
```

```
            ("High Load", {"rate": 1000, "duration": 180}),
            ("Spike Test", {"rate": 100, "duration": 300, "test_type":
"spike",
                            "spike_rate": 2000, "spike_duration": 60}),
        ]

        try:
            from advanced_load_test import LoadTester

            for scenario_name, config in test_scenarios:
                print(f"\n{'='*50}")
                print(f"Starting: {scenario_name}")
                print(f"{'='*50}")

                # Wait between tests
                if self.test_results:
                    print("Cooling down for 30 seconds...")
                    time.sleep(30)

                async with LoadTester(self.api_url,
config.get("connections", 1000)) as tester:
                    if config.get("test_type") == "spike":
                        await tester.run_spike_test(
                            config["spike_rate"],
                            config["spike_duration"],
                            config["rate"],
                            config["duration"]
                        )
                    else:
                        await
tester.run_constant_rate_test(config["rate"], config["duration"])

                    # Store results
                    self.test_results.append({
                        "scenario": scenario_name,
                        "config": config,
                        "metrics": tester.metrics.get_summary(),
                        "system_stats":
self.monitor.get_current_stats()
                    })

                print(f"{scenario_name} completed")

        finally:
            # Stop monitoring
            self.monitor.stop_monitoring()

        # Generate comprehensive report
        self._generate_comprehensive_report()
```

```
    def _generate_comprehensive_report(self):
        """Generate comprehensive test report"""
        timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
        report_file =
f"comprehensive_load_test_report_{timestamp}.json"

        report = {
            "test_suite": "Comprehensive Load Test",
            "timestamp": timestamp,
            "api_url": self.api_url,
            "scenarios": self.test_results,
            "summary": self._calculate_summary_stats()
        }

        with open(report_file, 'w') as f:
            json.dump(report, f, indent=2, default=str)

        print(f"\nComprehensive report saved to: {report_file}")

        # Generate plots
        plot_file = f"performance_metrics_{timestamp}.png"
        self.monitor.plot_metrics(plot_file)

        # Print summary
        self._print_summary_report()

    def _calculate_summary_stats(self) -> dict:
        """Calculate summary statistics across all tests"""
        if not self.test_results:
            return {}

        all_latencies = []
        total_requests = 0
        total_successful = 0
        max_rps = 0

        for result in self.test_results:
            metrics = result["metrics"]
            total_requests += metrics.get("total_requests", 0)
            total_successful += metrics.get("successful_requests", 0)
            max_rps = max(max_rps, metrics.get("requests_per_second",
0))

            # Collect latency data
            if "percentiles" in metrics:
                all_latencies.extend([metrics["average_latency_ms"]])

        return {
            "total_requests_across_all_tests": total_requests,
            "overall_success_rate": (total_successful /
```

```
max(total_requests, 1)) * 100,
            "peak_requests_per_second": max_rps,
            "average_latency_across_tests": sum(all_latencies) /
len(all_latencies) if all_latencies else 0
        }

    def _print_summary_report(self):
        """Print summary of all test results"""
        print("\n" + "="*80)
        print("COMPREHENSIVE LOAD TEST SUMMARY")
        print("="*80)

        for i, result in enumerate(self.test_results):
            scenario = result["scenario"]
            metrics = result["metrics"]

            print(f"\n{i+1}. {scenario}")
            print("-" * (len(scenario) + 3))
            print(f"Requests/sec: {metrics.get('requests_per_second',
0):,.2f}")
            print(f"Success Rate: {metrics.get('success_rate', 0):.2f}
%")
            print(f"Avg Latency: {metrics.get('average_latency_ms',
0):.2f}ms")
            print(f"P99 Latency: {metrics.get('percentiles',
{}).get('p99', 0):.2f}ms")
            print(f"Orders: {metrics.get('orders_submitted', 0):,}")
            print(f"Trades: {metrics.get('trades_executed', 0):,}")

        summary = self._calculate_summary_stats()
        print(f"\nOVERALL SUMMARY:")
        print(f"Total Requests:
{summary.get('total_requests_across_all_tests', 0):,}")
        print(f"Overall Success Rate:
{summary.get('overall_success_rate', 0):.2f}%")
        print(f"Peak RPS: {summary.get('peak_requests_per_second',
0):,.2f}")
        print("="*80)


# Simple standalone scripts for quick testing

def quick_burst_test(url: str = "http://localhost:5000", concurrent:
int = 100, requests: int = 1000):
    """Quick burst test using threading"""
    import threading
    import requests
    import time
    from collections import deque
```

```
    results = deque()

    def make_request():
        try:
            start = time.time()
            response = requests.post(
                f"{url}/order",
                json={
                    "symbol": "BTC-USDT",
                    "side": "buy",
                    "order_type": "limit",
                    "quantity": "0.001",
                    "price": "29000"
                },
                timeout=10
            )
            duration = (time.time() - start) * 1000
            results.append({
                "success": response.status_code == 200,
                "duration_ms": duration,
                "status": response.status_code
            })
        except Exception as e:
            results.append({
                "success": False,
                "duration_ms": -1,
                "error": str(e)
            })

    print(f"Starting burst test: {requests} requests with {concurrent}
concurrent threads")
    start_time = time.time()

    # Create and start threads
    threads = []
    requests_per_thread = requests // concurrent

    for _ in range(concurrent):
        for _ in range(requests_per_thread):
            t = threading.Thread(target=make_request)
            threads.append(t)

    # Start all threads
    for t in threads:
        t.start()

    # Wait for completion
    for t in threads:
        t.join()
```

```
    total_time = time.time() - start_time

    # Analyze results
    successful = sum(1 for r in results if r["success"])
    failed = len(results) - successful
    valid_durations = [r["duration_ms"] for r in results if
r["duration_ms"] > 0]

    print(f"\nBURST TEST RESULTS:")
    print(f"Total Time: {total_time:.2f}s")
    print(f"Requests: {len(results)}")
    print(f"Successful: {successful}
({successful/len(results)*100:.2f}%)")
    print(f"Failed: {failed}")
    print(f"RPS: {len(results)/total_time:.2f}")

    if valid_durations:
        print(f"Avg Latency:
{sum(valid_durations)/len(valid_durations):.2f}ms")
        print(f"Min Latency: {min(valid_durations):.2f}ms")
        print(f"Max Latency: {max(valid_durations):.2f}ms")


def profile_single_request(url: str = "http://localhost:5000",
iterations: int = 100):
    """Profile single request performance"""
    import requests
    import time
    import statistics

    durations = []

    print(f"Profiling single requests ({iterations} iterations)...")

    for i in range(iterations):
        try:
            start = time.time()
            response = requests.post(
                f"{url}/order",
                json={
                    "symbol": "BTC-USDT",
                    "side": "buy",
                    "order_type": "limit",
                    "quantity": "0.001",
                    "price": str(29000 + i)  # Vary price slightly
                },
                timeout=5
            )
            duration = (time.time() - start) * 1000
```

```
            if response.status_code == 200:
                durations.append(duration)

            if (i + 1) % 10 == 0:
                print(f"Completed {i + 1}/{iterations} requests...")

        except Exception as e:
            print(f"Request {i+1} failed: {e}")

    if durations:
        print(f"\nSINGLE REQUEST PROFILE:")
        print(f"Successful requests: {len(durations)}")
        print(f"Average latency: {statistics.mean(durations):.2f}ms")
        print(f"Median latency: {statistics.median(durations):.2f}ms")
        print(f"Min latency: {min(durations):.2f}ms")
        print(f"Max latency: {max(durations):.2f}ms")
        print(f"Std deviation: {statistics.stdev(durations):.2f}ms")


if __name__ == "__main__":
    import argparse
    import asyncio

    parser = argparse.ArgumentParser(description="Monitoring and
Testing Suite")
    parser.add_argument("--url", default="http://localhost:5000",
help="API URL")
    parser.add_argument("--test", choices=["monitor", "comprehensive",
"burst", "profile"],
                        default="monitor", help="Test type to run")
    parser.add_argument("--concurrent", type=int, default=100,
help="Concurrent requests for burst test")
    parser.add_argument("--requests", type=int, default=1000,
help="Total requests for burst test")
    parser.add_argument("--iterations", type=int, default=100,
help="Iterations for profile test")

    args = parser.parse_args()

    if args.test == "monitor":
        monitor = SystemMonitor(args.url)
        try:
            monitor.start_monitoring()
            print("Monitoring system... Press Ctrl+C to stop and
generate report")
            while True:
                stats = monitor.get_current_stats()
                if stats:
                    print(f"CPU: {stats.get('cpu_percent', 0):.1f}% |
"
```

```
                            f"Memory: {stats.get('memory_percent',
0):.1f}% | "
                            f"API: {stats.get('api_response_time_ms', -
1):.1f}ms")
                time.sleep(5)
        except KeyboardInterrupt:
            monitor.stop_monitoring()
            monitor.plot_metrics()

    elif args.test == "comprehensive":
        orchestrator = LoadTestOrchestrator(args.url)
        asyncio.run(orchestrator.run_comprehensive_test_suite())

    elif args.test == "burst":
        quick_burst_test(args.url, args.concurrent, args.requests)

    elif args.test == "profile":
        profile_single_request(args.url, args.iterations)
```