Holly AI to IBKR Bridge - Detailed Implementation Specification

1. System Overview

Purpose

Automated trading system that reads Holly AI breakout signals from daily CSV files and executes trades through Interactive Brokers TWS API with sophisticated risk management and time-based exits.

Core Requirements

- **Signal Source**: Daily CSV files from Trade Ideas Holly AI
- **Execution**: Interactive Brokers TWS API (ib_insync)
- **Risk Management**: 3% position sizing, max 3 concurrent positions, 30 daily trades
- **Exit Strategy**: 1% stop loss with 10-minute time-based exit (no profit targets)
- **Reliability**: Must handle disconnections, errors, and maintain position synchronization

2. Architecture

Component Structure

```
src/
├── main.py              # Application entry point and orchestration
├── core/
│   ├── ibkr_connector.py   # IBKR API wrapper with connection management
│   ├── order_manager.py    # Order lifecycle and execution
│   ├── risk_manager.py     # Position sizing and risk rules
│   └── position_tracker.py # Position state synchronization (NEW)
├── utils/
│   ├── csv_parser.py       # Holly signal parsing
│   ├── logger.py           # Centralized logging
│   └── config_loader.py    # Configuration management
└── monitors/
    ├── time_exit_monitor.py # Time-based exit thread (NEW)
    └── health_monitor.py    # System health checks (NEW)
```

Data Flow

```
Holly CSV → Parser → Risk Manager → Order Manager → IBKR
                                          ↓
                                  Position Tracker ← IBKR Positions
                                          ↓
                                  Time Exit Monitor
```

3. Critical Implementation Details

3.1 Position State Management

Problem: Current system loses sync between tracked positions and actual IBKR positions.

Solution: Implement a `PositionTracker` that maintains authoritative position state:

```python
```

```python
class PositionTracker:
    """
    Single source of truth for position state.
    Syncs with IBKR on startup and periodically.
    """

    def __init__(self, ib_connector):
        self.ib = ib_connector
        self.positions = {}  # symbol -> PositionState
        self.pending_exits = {}  # symbol -> exit_time

    def sync_with_ibkr(self):
        """
        Compare internal state with IBKR positions.
        Resolve discrepancies with IBKR as authoritative source.
        """

    def add_position(self, symbol, shares, entry_price, order_id):
        """
        Add new position with all metadata.
        Store parent_order_id for bracket order management.
        """

    def schedule_time_exit(self, symbol, exit_time):
        """
        Schedule time-based exit.
        Store all order IDs that need cancellation.
        """
```

3.2 Order Lifecycle Management

Problem: Orders fail to transmit, get lost, or aren't properly tracked.

Solution: Implement comprehensive order state machine:

```python
```

```
class OrderState(Enum):
    PENDING = "pending"
    SUBMITTED = "submitted"
    FILLED = "filled"
    CANCELLED = "cancelled"
    FAILED = "failed"

class OrderLifecycle:
    """
    Track every order from creation to completion.
    Handle all edge cases and error scenarios.
    """

    def place_entry_with_stop(self, symbol, shares, stop_price):
        """
        1. Create and qualify contract
        2. Place parent market order with transmit=False
        3. Place stop order with parentId
        4. Transmit both orders
        5. Verify submission status
        6. Return order IDs for tracking
        """

    def execute_time_exit(self, symbol, position_state):
        """
        1. Cancel all child orders (stop/limit)
        2. Verify cancellation completed
        3. Place market exit order
        4. Verify order transmitted
        5. Monitor fill status
        6. Handle partial fills
        """
```

3.3 Time-Based Exit Implementation

Problem: Current implementation fails to reliably execute exits.

Solution: Dedicated monitor with robust execution:

```
python
```

```python
class TimeExitMonitor(Thread):
    """
    Dedicated thread for time-based exits.
    Runs independently with its own error handling.
    """

    def __init__(self, position_tracker, order_manager):
        self.tracker = position_tracker
        self.order_mgr = order_manager
        self.running = True
        self.check_interval = 5  # seconds

    def run(self):
        while self.running:
            try:
                positions_to_exit = self.tracker.get_due_exits()

                for symbol, position in positions_to_exit:
                    self.execute_exit(symbol, position)

            except Exception as e:
                logger.error(f"Exit monitor error: {e}")
                # Continue running - don't crash thread

            time.sleep(self.check_interval)

    def execute_exit(self, symbol, position):
        """
        1. Log exit attempt
        2. Cancel child orders with verification
        3. Place exit order with retries
        4. Update position tracker
        5. Log completion or failure
        """
```

3.4 Connection and Error Recovery

Problem: System doesn't handle disconnections or errors gracefully.

Solution: Implement connection manager with auto-recovery:

```python
```

```
class ConnectionManager:
    """
    Manage IBKR connection with automatic recovery.
    """

    def __init__(self, config):
        self.config = config
        self.ib = None
        self.connected = False
        self.reconnect_attempts = 0
        self.max_reconnect_attempts = 5

    def ensure_connected(self):
        """
        Check connection health and reconnect if needed.
        Called before any IBKR operation.
        """
        if not self.is_healthy():
            self.reconnect()

    def is_healthy(self):
        """
        1. Check ib.isConnected()
        2. Test with simple request (reqCurrentTime)
        3. Verify response within timeout
        """

    def reconnect(self):
        """
        1. Disconnect cleanly if connected
        2. Wait with exponential backoff
        3. Attempt new connection
        4. Resync positions after connection
        5. Raise if max attempts exceeded
        """
```

3.5 Risk Management Enhancement

Problem: Risk checks need to be more comprehensive.

Solution: Enhanced risk manager with state validation:

```
python
```

```
class RiskManager:
    """
    Enforce risk rules with comprehensive validation.
    """

    def __init__(self, config, position_tracker):
        self.config = config
        self.tracker = position_tracker
        self.daily_trades = 0
        self.last_reset = date.today()

    def validate_new_trade(self, signal):
        """
        1. Reset daily counters if new day
        2. Check concurrent positions (authoritative from tracker)
        3. Check daily trade limit
        4. Check duplicate symbol
        5. Validate account buying power
        6. Return detailed rejection reason if failed
        """

    def calculate_position_size(self, price, account_value):
        """
        1. Get current account value from IBKR
        2. Calculate 3% of account
        3. Calculate shares (round down)
        4. Verify doesn't exceed buying power
        5. Return shares and dollar amount
        """
```

3.6 CSV Parser Robustness

Problem: Parser fails on column mismatches or file issues.

Solution: Adaptive parser with validation:

```
python
```

```
class CSVParser:
    """
    Robust CSV parser with auto-detection and validation.
    """

    def __init__(self, config):
        self.config = config
        self.column_mappings = {}
        self.validated = False

    def parse_daily_file(self):
        """
        1. Find today's file with date pattern
        2. Validate file exists and is readable
        3. Auto-detect columns on first run
        4. Parse with error handling per row
        5. Return valid signals only
        """

    def auto_detect_columns(self, df):
        """
        1. Get actual column names
        2. Match against common patterns
        3. Validate required columns present
        4. Store mapping for future use
        5. Log detected configuration
        """
```

4. Critical Workflows

4.1 Startup Sequence

python

```
def startup_sequence():
    """
    1. Load and validate configuration
    2. Initialize logging
    3. Connect to IBKR with retry
    4. Sync positions with IBKR
    5. Recover any pending exits
    6. Start monitors (time exit, health)
    7. Begin signal processing
    """
```

4.2 Signal Processing Flow

python

```
def process_signal(signal):
    """
    1. Parse signal from CSV
    2. Risk validation checks
    3. Calculate position size
    4. Place entry order with stop
    5. Verify order accepted
    6. Update position tracker
    7. Schedule time exit
    8. Log all actions
    """
```

4.3 Time Exit Execution

python

```
def execute_time_exit(symbol, position):
    """
    1. Log exit initiation
    2. Get current position from tracker
    3. Cancel stop order (with verification)
    4. Wait for cancellation confirmation
    5. Place market exit order
    6. Monitor order status
    7. Verify fill
    8. Update position tracker
    9. Log completion
    """
```

4.4 Graceful Shutdown

python

```
def shutdown_sequence():
    """
    1. Stop accepting new signals
    2. Close all open positions
    3. Wait for order completion
    4. Stop all monitors
    5. Disconnect from IBKR
    6. Final position sync check
    7. Log shutdown stats
    """
```

5. Error Handling Patterns

5.1 Order Errors

python

```
def handle_order_error(error, order, retry_count=0):
    """
    Error codes to handle:
    - 201: Order rejected - Invalid stop price
    - 202: Order cancelled
    - 2106: HMDS data farm connection is OK
    - Connection lost errors

    Actions:
    1. Log full error details
    2. Determine if retryable
    3. Update position tracker
    4. Notify monitors
    5. Execute fallback if needed
    """
```

5.2 Connection Errors

```
python

def handle_connection_error(error):
    """
    1. Log disconnection
    2. Attempt immediate reconnect
    3. If failed, exponential backoff
    4. Sync positions after reconnect
    5. Resume normal operation
    6. Alert if extended downtime
    """
```

6. State Persistence

6.1 Position State File

```
json
```

```
{
  "positions": {
    "AAPL": {
      "shares": 42,
      "entry_price": 70.80,
      "entry_time": "2024-07-25T20:40:46",
      "scheduled_exit": "2024-07-25T20:50:46",
      "parent_order_id": 9,
      "stop_order_id": 10,
      "status": "open"
    }
  },
  "daily_stats": {
    "date": "2024-07-25",
    "trades_taken": 1,
    "trades_remaining": 29
  }
}
```


6.2 Recovery on Restart

```python
def recover_state():
    """
    1. Load persisted state
    2. Sync with IBKR positions
    3. Reconcile differences
    4. Resume exit schedules
    5. Update daily counters
    """
```

7. Monitoring and Alerting

7.1 Health Checks

```python
```

```python
class HealthMonitor:
    """
    Monitor system health and alert on issues.
    """

    checks = [
        "ibkr_connection",
        "position_sync",
        "order_execution_rate",
        "csv_file_availability",
        "time_exit_success_rate"
    ]

    def run_health_checks(self):
        """
        Run all checks every 60 seconds.
        Log warnings for degraded state.
        Send alerts for critical issues.
        """
```

7.2 Metrics to Track

- Orders placed/filled/cancelled
- Position sync mismatches
- Time exit success rate
- Connection uptime
- Daily P&L
- Risk rule violations

8. Testing Strategy

8.1 Unit Tests

python

```
def test_position_sizing():
    """Test 3% position calculation with various account values"""

def test_risk_validation():
    """Test all risk rule scenarios"""

def test_time_exit_scheduling():
    """Test exit time calculations"""
```

8.2 Integration Tests

python

```
def test_full_trade_lifecycle():
    """
    1. Mock CSV signal
    2. Process through system
    3. Verify IBKR orders
    4. Trigger time exit
    5. Verify position closed
    """
```

8.3 Paper Trading Validation

- Run for 2 weeks minimum
- Compare actual fills vs expected
- Validate time exits execute properly
- Check position sync accuracy
- Monitor error rates

9. Configuration Schema

yaml

```yaml
# config.yaml
alerts:
  csv_path: "data/alerts/"
  strategy_name: "Breaking out on Volume"
  file_prefix: "alertlogging"
  check_interval: 1
  auto_detect_columns: true

risk_management:
  max_daily_trades: 30
  max_concurrent_positions: 3
  position_size_pct: 3.0
  stop_loss_pct: 1.0
  time_exit_minutes: 10

ibkr:
  host: "127.0.0.1"
  port: 7497
  client_id: 1
  connection_timeout: 30
  order_timeout: 10

monitoring:
  health_check_interval: 60
  position_sync_interval: 300

recovery:
  max_reconnect_attempts: 5
  reconnect_delay_base: 2
  state_file: "data/position_state.json"
```

10. Production Readiness Checklist

- [ ] All orders have proper error handling
- [ ] Position state persists across restarts
- [ ] Automatic IBKR reconnection works
- [ ] Time exits execute reliably
- [ ] Position sync detects mismatches
- [ ] Graceful shutdown closes all positions
- [ ] Health monitoring alerts on issues
- [ ] Logs capture all critical events

- [ ] Configuration is validated on startup
- [ ] Paper trading for 2+ weeks successful

11. Known Edge Cases to Handle

1. **Partial Fills**: Handle when order fills partially
2. **After-Hours**: Disable trading outside market hours
3. **Halted Stocks**: Detect and skip halted symbols
4. **Connection Loss During Exit**: Ensure position still closes
5. **Duplicate Signals**: Prevent double orders on same signal
6. **Stop Order Rejection**: Have fallback exit strategy
7. **Account Restrictions**: Handle PDT and buying power limits

12. Future Enhancements

1. **Multiple Strategies**: Support different Holly strategies
2. **Dynamic Position Sizing**: Adjust based on volatility
3. **Performance Analytics**: Track strategy metrics
4. **Web Dashboard**: Real-time monitoring interface
5. **Mobile Alerts**: Push notifications for issues
6. **Strategy Optimization**: A/B test parameters

This specification provides the blueprint for a production-grade trading system that addresses all discovered issues and implements best practices for reliability and safety.